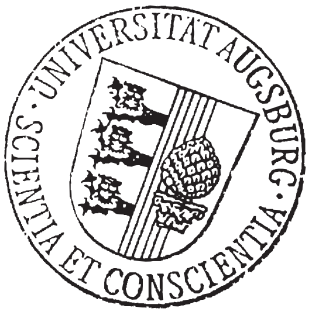
Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors

Dissertation

zur
Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

der Fakultät für angewandte Informatik
der Universität Augsburg



Universität
Augsburg
University

eingereicht von:

Dipl.-Inf. Jan Nowotsch

geb. am 14.06.1987 Karl-Marx-Stadt

Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors
Jan Nowotsch

Erstgutachter: Prof. Dr. Theo Ungerer
Zweitgutachter: Prof. Dr. Rudi Knorr

Datum der mündlichen Prüfung: 16.05.2014

Abstract

Timing matters. This is especially true for safety-critical real-time applications, since human lives depend on their correctness. Such applications are naturally used in the avionics and automotive industries. Within these domains, the decreasing relative costs and the pace of micro-electronics development have led to the adoption of Commercial Off-The-Shelf (COTS) components in recent years. Likewise, multi-core processors are an interesting alternative to established single-core systems. Unfortunately, multi-core architectures pose significant challenges with respect to analysability and predictability, impeding the proof of temporal correctness and thus system safety. In particular, the inherent usage of shared resources induce dependencies between processor cores. As a consequence instruction latencies on one core can be influenced by the code executed on other cores. This constitutes an inter-core dependency as it does not exist for single-core processors.

This thesis addresses the problems of worst-case timing analysis in the presence of inter-core interferences due to the implicit use of shared resources on multi-core processors. Additionally, a mechanism that enables an efficient utilisation of resources to increase the average-case performance is proposed. In terms of timing analysis the concept of interference-sensitive Worst-Case Execution Time (isWCET) is proposed. It is based on an extended timing analysis to account for resource interferences and a runtime resource usage enforcement to ensure hard deadline guarantees. The offline analysis is used to determine upper bounds on execution times and resource usage behaviour to define a static system configuration. On this basis, runtime resource usage enforcement ensures that the assumptions made during analysis are met at runtime. Besides guaranteed worst-case behaviour the average-case performance is a crucial factor to benefit over established single-core processors. Hence, it is essential to efficiently utilise the parallel architecture features of multi-core systems. Having this in mind, a Quality of Service (QoS) extension to the isWCET concept is implemented. It enables the dynamic re-computation of the former static configuration, based on the actual resource usage of applications during execution.

The evaluation of the isWCET concept and its QoS extension is based on the Freescale P4080, a state of the art multi-core processor, and AbsInt's well known timing analysis framework aiT. The results reveal a reduction of the multi-core timing bound of up to 86.3% compared to a straight forward approach. Further, the dynamic re-computation enables an increase of the processor core utilisation from 0.5% to 99.9%, achieving a total system utilisation of up to 64.2%, compared to 9.5% otherwise. The overall results prove the validity and the benefits of the isWCET concept and its QoS extension. Furthermore, the evaluation illustrates the independence of the approaches from underlying hardware, software and applied analysis techniques. The comparison of the isWCET analysis to related approaches clearly illustrates its advantages. The isWCET concept allows the independent analysis of in-parallel scheduled applications, enabling incremental development and certification. Further, the approaches allow the concurrent use of shared resources without requiring per se resource privatisation and modifications to the applications or the underlying hardware. Also, priorisation between applications can be avoided, which eases the application mixed-criticality systems. Accordingly, the isWCET approach is considered a promising alternative for deploying commercial multi-core processors in future safety-critical, hard real-time systems.

Kurzzusammenfassung

Pünktlichkeit ist wichtig. Dies trifft besonders auf Anwendungen im Bereich sicherheitskritischer Echtzeitsysteme zu, da Menschenleben von ihrer Funktionstüchtigkeit abhängen. Entsprechende Anwendungen sind zum Beispiel in der Avionik- und Automobilindustrie zu finden. In diesen Zweigen hat der rapide Fortschritt in der Mikroelektronik und die gesunkenen relativen Kosten über die letzten Jahre zum vermehrten Einsatz von Fertigbauteilen, sogenannten COTS Komponenten geführt. Als ein nächster Schritt werden Multi-core Prozessoren als Alternativen zu etablierten Single-core Prozessoren in Betracht gezogen. Im Gegensatz zu Single-core Prozessoren ergeben sich durch den Einsatz von Multi-core Prozessoren jedoch Probleme im Hinblick auf Vorhersagbarkeit und Analysierbarkeit, was den Nachweis zeitlicher Korrektheit und damit der Systemsicherheit deutlich erschwert. Das Hauptproblem hierbei sind inhärente Abhängigkeiten zwischen Prozessorkernen, die durch die gemeinsame Nutzung von Ressourcen entstehen. Als Folge dessen können die Ausführungszeiten einzelner Instruktionen auf einem Kern durch die auf anderen Kernen ausgeführten Programme beeinflusst werden. Damit ergibt sich eine Abhängigkeit wie sie für Single-core Prozessoren nicht gegeben ist.

In der vorliegenden Arbeit wird das Problem der Laufzeitanalyse von Multi-core Prozessoren unter Berücksichtigung von zeitlichen Abhängigkeiten zwischen Prozessorkernen behandelt. Außerdem wird untersucht, wie im Kontext harter Echtzeitanforderungen eine effiziente Auslastung des Gesamtsystems erreicht werden kann. Hierzu wird das Konzept der Interferenz-sensitiven Laufzeitanalyse (isWCET) vorgestellt. Das grundlegende isWCET Konzept basiert auf einer erweiterten Laufzeitanalyse, welche zunächst die Ressourcennutzung einzelner Anwendungen bestimmt und auf deren Basis mögliche Interferenzen bei der Berechnung oberer Schranken für die Gesamtlaufzeit berücksichtigt. Aus den bestimmten Laufzeit- und Ressourceninformationen wird eine zunächst statische Systemkonfiguration erstellt. Diese wird während der Ausführung von einem, im Rahmen der Arbeit entwickelten, Kontrollmechanismus überwacht. Damit wird sichergestellt, dass alle Annahmen, die zur Analyse notwendig sind, während der Ausführung eingehalten werden. Zusätzlich zum isWCET Konzept wird eine QoS Erweiterung vorgestellt, welche eine dynamische Anpassung der Systemkonfiguration auf Basis des tatsächliches Systemfortschritts erlaubt.

Mit der Evaluierung werden schließlich die Vorteile, die durch den Einsatz der vorgestellten Konzepte entstehen ermittelt. Die berechneten Schranken für Programmlaufzeiten können um bis zu 86.3% reduziert werden, während die Prozessorkernauslastung maximal von 0.5% auf 99.9% erhöht werden kann. Ebenso kann die Gesamtauslastung des Systems von 9.5% auf 64.2% gesteigert werden. Damit können sowohl die Vorteile, wie auch die Notwendigkeit der vorgestellten Lösungen unter Beweis gestellt werden. Neben den quantitativen Verbesserungen werden außerdem die Unabhängigkeit der Konzepte von eingesetzter Hard- und Software, sowie Analysetechnik gezeigt. Im Vergleich zu verwandten Arbeiten sind die Hauptunterschiede die Fähigkeit zur unabhängigen Analyse von parallel ausgeführten Anwendungen und die Vermeidung von Ressourcenprivatisierung. Damit werden die Anforderungen von inkrementeller Entwicklung und Zertifizierung erfüllt und eine effizientere Nutzung von Ressourcen, als es bei einigen bekannten Ansätzen möglich ist, erreicht. Außerdem wird nicht zwischen verschiedenen Anwendungen priorisiert, was den Einsatz in gemischt kritischen Systemen erlaubt. Damit stellen die Konzepte dieser Arbeit eine interessante Alternative zu bestehenden Lösungen dar.

Contents

1. Introduction

1.1. Motivation

Safety-critical systems are used in domains such as avionics, automotive and industrial control, where human lives depend on their correct operation. Correct operation is often ensured via certification. Whereat, depending on the particular safety level applications need to be analysed in various details. Often, the implementation of a specific system is split into smaller applications. In order to form a complete system all applications need to be integrated. The temporal parametrisation usually relies on their worst-case performance in order to ensure safe operation even in exceptional situations. In consequence, safety-critical systems are also real-time systems. While a system consists of multiple applications, different applications may be implemented independently from each other. This also includes, that modifications to one application shall not impact the behaviour and analysis of other, unmodified applications. This concept is also called incremental certification and development [Wilson and Preyssler (2008), RTCA (2005)]. In this context, the term partitioning is commonly used to refer to the runtime separation of applications. Partitioning comprises spatial and temporal aspects, including the separation of address spaces and bounding of timing influences between applications. Therefore, all applied tools, such as the timing analysis and also the runtime environment have to support these modular processes.

In recent years the development of safety-critical systems showed a trend towards increased integration coupled with higher performance demands [Triquet (2012)]. The transition from host processors towards System-on-Chip (SoC) designs increased system integration, while the deployment of Commercial Off-The-Shelf (COTS) components allowed significant performance advantages. Considering further growing performance needs of applications and the integration of more advanced functions, multi-core processors are the logical next evolutionary step [Reichenbach and Wold (2010), EC (2012), Duranton et al. (2011)]. Unfortunately, SoC and multi-core designs as well as the use of commercial components pose new and very different challenges with respect to safe partitioning and certification [EASA (2011), Dasari et al. (2013), Fuchsen (2010), Kinnan (2009)]. COTS processors are designed for average-case performance, e.g. integrating complex execution pipelines and multi-level cache hierarchies, which often contradicts predictability. Accordingly, the EASA classified COTS components into simple, complex and highly complex, depending on the feature set of the device [EASA (2011)]. Multi-core processors are considered as highly complex due to the use of multiple processing units. Further, the design documentation of COTS hardware often omits relevant implementation details to maintain the competitive advantages of the vendors. For instance, design details of the interconnection between processing units, main memory and peripheral interfaces are keep secret, since significant development efforts are spent in their design. Unfortunately, detailed documentation is typically required for analysis and certification [EASA (2011)]. The main concern with respect to SoC-designs is the high integration level of the system. Usually, SoC platforms integrate a processor core, memory controllers and peripheral devices within the same chip. This introduces additional sources of faults and potential interactions between units. Beneath others, this increases the effort and complexity of certification compared to host processor designs. Finally, multi-core architectures integrate multiple processing units within the same chip, competing for shared resources and ultimately influencing each others

temporal behaviour. In effect, the latencies of individual resource requests do no longer solely depend on the analysed application. Instead, mutual interactions between applications running on different cores need to be considered during timing analysis.

For single-core systems different processor contexts can be analysed, since context changes are caused by the analysed application. Unknown context information, e.g. due to unknown initial states, are usually covered by assumptions during analysis and respective runtime enforcements. For example, the analysis might assume empty caches as the initial state, while the operating system ensures this via cache flushes at each task switch. Single-core timing analysis is a research topic since the late 1980s [Shaw (1989), Puschner and Koza (1989)]. Popular approaches are static analysis and hybrid measurement-based approaches, cf. [Wilhelm et al. (2008)]. The analysis precision constantly improved, from relatively simple architectures [Park and Shaw (1990), Puschner and Schedl (1995), Puschner and Schedl (1997)] to complex architectures that include features such as pipelines [Healy et al. (1999), Stappert et al. (2001)], branch predictors [Colin and Puaut (2001), Li et al. (2005), Mitra et al. (2002), Colin and Puaut (2000)] and caches [Healy et al. (1999), Arnold et al. (1994), Ramaprasad and Mueller (2005)].

Research of multi-core timing analysis is driven towards two directions firstly, the analysis of parallelised applications and secondly, the analysis of multiple independent single-core applications scheduled on different cores. Likewise, different challenges are imposed. A focal point for parallelised applications are synchronisation operations between parallel instances, as for instance discussed in [Gerdes et al. (2012a), Gerdes et al. (2012b)]. The parallelisation shall not only reduce the average-case, but also the worst-case execution time. On the other hand, the main challenge for the analysis of independent applications scheduled on multi-core processors is the analysis of shared resource interferences, as described above. In contrast to parallelised applications, the application code is not optimised for multi-core systems. Instead inter-application interference increase instruction latencies in the worst-case. Hence, it is not expected to reduce the Worst-Case Execution Time (WCET) bounds compared to single-core systems. In literature four main approaches towards the analysis of independent applications on multi-core processors are proposed: the application of execution models, joint analysis, analysed/controlled sharing and dedicated hardware designs. The main drawbacks of these approaches are infeasible complexity and scalability, assumptions that contradict the requirements of incremental development and certification, per se resource privatisation, priorisation between applications contradicting the requirements of mixed-criticality systems and the inapplicability of dedicated hardware for COTS systems.

In summary, the problems related to COTS components and SoC-designs mainly increase the effort and complexity of certification. On the other hand, concurrency issues due to multi-core designs impact the predictability, which in turn affects temporal correctness and system safety. As a consequence, the analysis of resource interferences is a crucial challenge for the deployment of multi-core processors for real-time systems. Even though some approaches already address this challenge, they do not sufficiently solve it. Accordingly, an alternative method to address the analysis of inter-application interferences due to the resource sharing in multi-core systems is proposed in this thesis. Therefor, the implications of COTS components, SoC-designs and the requirements of safety-critical systems towards incremental development and certification and mixed-criticality are considered.

1.2. Contributions

Within the thesis spatial partitioning is considered to be sufficiently solved for the deployment of multi-core processors to safety-critical systems. Especially, since features such as Memory Management Units (MMUs) and Input/Output MMUs (I/OMMUs) are not solely required by safety-critical applications. Consequently, it is an objective of this work to implement a temporal partitioning mechanism. An essential aspect of temporal partitioning is the worst-case timing and resource analysis. Hence, the main focus is on bounding the WCET, considering non-determinism in access delays due to the concurrent use of shared resources. Additional objectives are to allow incremental analysis of applications without modifying the application binaries, requiring resource privatisation or restricting the freedom of developers by additional coding guidelines. Furthermore, all concepts shall be functionally and temporally transparent to applications. Meaning, neither applications shall be able to control the partitioning nor suffer timing influences due to its existence.

To achieve these objective, the concept of the so-called *interference-sensitive Worst-Case Execution Time (isWCET)* analysis is introduced. It is targeted to compute execution time bounds for independent applications scheduled on multi-core processors. Intuitively explained, the isWCET concept separates core-local and shared resource timing analyses. The behaviour of applications is abstracted as bounds for their WCET and resource usage. The non-determinism in access delays is expressed as inter-application interferences, which are computed as the combination of the resource analysis results of all in-parallel scheduled applications. The final isWCET bound of an application consists of its core-local timing and the maximum delay due to inter-application interferences. The offline analysis is complemented by runtime monitoring to enforce the computed resource usage boundaries and ensure, that assumptions made during analysis are met during execution.

While determining the worst-case behaviour of applications is essential to parametrise the system, average-case performance is necessary to efficiently utilise the available resources. Therefore, the basic isWCET concept is extended to determine the unused resources and assign them to demanding applications. For that purpose, the *capacity-extension* concept is introduced. It allows dynamic re-computations of the resource usage bounds, depending on the progress of in-parallel scheduled applications. The concept is designed such, that timing and resource constraints of other applications are not violated in order to retain hard real-time guarantees.

The proposed concepts are based on a generic multi-core architecture model, abstracting a shared-memory system. The evaluation is based on the P4080, Freescale's reference architecture for the QorIQ Multi-Processor System-on-Chip (MPSoC) processor family. While the presented concepts are generally valid for all kinds of shared resources, the implementation and evaluation are targeted at shared main memories and the corresponding interconnect. To evaluate the basic isWCET concept the effects on the worst-case timing bounds of applications are quantified. For the evaluation of the capacity extension, the utilisation of the system and of individual cores with and without the extension are investigated. Also the number of additional resource requests is analysed. The core-local timing analysis is implemented based on the static timing analysis framework, aiT from AbsInt, while the evaluation proves the applicability of measurement-based approaches as well.

This thesis extends the state of the art in WCET analysis for multi-core processors. The main contributions are outlined in the following:

I The proposed **isWCET** timing analysis method allows the computation of timing bounds for applications that are scheduled in parallel on multi-core processors. Existing timing analysis techniques are extended to additionally analyse the resource usage of applications. The maximum interference on shared resources can be bounded, combining the

information on the resource usage of all applications. In contrast to related approaches, interferences are bounded without requiring changes to the applications or the target hardware. Further, instead of relying on per se resource privatisation, the use of parallel architecture features, such as multi-channel interconnects, is enabled, contrary to enforcing resource privatisation to avoid any interferences. In consequence, the proposed concept enables incremental development and certification, which is essential for industrial applicability. Compared to commonly applied maximum-contention approaches, the isWCET analysis reduces the resulting timing bounds.

II Based on the offline timing analysis a **runtime resource usage enforcement** mechanism is proposed, that ensures temporal isolation of applications. This is essential to ensure, that assumptions which have been made during analysis are followed at runtime. Additionally, the mechanism provides a, so-called safety-net, which is an design alternative to ensure fault containment and increased system safety [Green et al. (2011)].

III Existing approaches for real-time systems often divide applications into critical and non-critical. If multiple applications are executed in parallel, there is only one critical application that is prioritized in case of conflicts. In contrast to such approaches, the proposed timing analysis and runtime enforcement ensure equal guarantees for timing and resource usage over all applications. This enables the integration of **mixed-criticality** task sets and in general the parallel execution of multiple hard real-time tasks, which is another essential property for industrial relevance.

IV Besides bounding the worst-case behaviour of applications this thesis also provides a mechanism to increase the processor core and system utilisation. This enables the efficient use of multi-core processors even if legacy applications are integrated without applying any parallelisation. Due to the additional interferences on shared resources multi-core WCET bounds of such unparallelised applications will always be greater or equal compared to their single-core counterparts. Nevertheless, the proposed **Quality of Service (QoS)** concept allows the efficient utilisation of multi-core processors, whereby multi-core systems can take advantage over single-core systems, while maintaining hard deadline guarantees.

In summary, beyond the quantitative improvements with respect to reduced timing bounds and increased system utilisation, the presented approaches shall maintain similar analysis complexity than existing single-core techniques, while considering the specific requirements of mixed-criticality systems and incremental development and certification.

1.3. Outline

The remainder of this thesis is structured as follows. In Chapter 2 essential background topics addressed. This includes the domain of safety-critical applications, which is introduced on the example of avionics concepts and standards. Further, since the partitioning concept is a commonly accepted concept to implement separation of applications in different domains, it is described in more details. Also, the state of the art in worst-case execution time analysis is summarised. The briefly mentioned related approaches are discussed in Chapter 3. This also includes discussions of their drawbacks with respect to the computation of multi-core execution time bounds and the separation of applications. The main concepts of the isWCET analysis and runtime resource usage enforcement are introduced in Chapter 4. Based on the concepts the implementation and evaluation are presented in Chapters 5 and 6. The results and identified benefits and weaknesses are discussed within Chapter 7. Chapter 8 summarises the thesis and outlines future work topics.

2. Background

The research area of safety-critical applications comprises different aspects, from top level system design down to the implementation of individual applications. In order to further motivate the work in this thesis the essential background areas are described in more details. Section 2.1 is used to clarify some notations that are used throughout this thesis. In Section 2.2 the implications of safety and certification on applications are described on the example of the aerospace domain, and more specifically the avionics, the electronics of an airplane. However, similar definitions and concepts are used in other safety-critical domains, e.g. automotive. The central concept of partitioning is presented in Section 2.3 and afterwards the basics and state of the art in WCET analysis are summarized in Section 2.4.

2.1. Terminology

Partition, Process and Thread: The notations of partition, process and thread have slightly different meanings depending on the domain and abstraction layer. For example, in General-Purpose Operating Systems (GPOSs) such as Linux, a *process* is a management structure, that, amongst others, defines a separate address space. This address space is shared between all of a processes' *threads* [Tanenbaum (2007)]. Whereas, a *thread* is the scheduling entity.

In today's avionics systems, implemented against the ARINC 653 interface [ARINC (2003)], the terms *partition* and *process* are defined. A *partition* defines an separate address space, similar to a GPOS process. Beyond that, temporal isolation between partition is guaranteed by separate time windows. The executable units within partitions are *processes*. Likewise GPOS threads, all processes of a partition share the same address space and time window [ARINC (2003)].

Throughout this thesis the notations of *partition* and *process* are used following the ARINC 653 standard. Whereas, the terms process and thread are used interchangeably to refer to an executable scheduling entity.

Avionics Function and Application: The terms avionics functions and applications are used interchangeably throughout this thesis to refer to a high-level system function, such as flight control or passenger entertainment. Each application consists of one or more partitions. It is important to note, that different applications might be implemented independently from each other, realising incremental development and certification [Wilson and Preyssler (2008), RTCA (2005)].

Network-on-Chip (NoC): To connect the processing cores of a multi-core processor and the additional Input/Output (I/O) units in SoC-based designs, a chip interconnect is required. In literature very different interconnect architectures are well known [Ungerer (1997)] and still under research, e.g. bus- [Walter et al. (2008), Udipi et al. (2010)], ring- [Deslauriers et al. (2006), Li-Guo et al. (2008)] and tree-based architectures [Matsutani et al. (2008), Matsutani et al. (2009)]. Within this thesis the concrete architecture of the interconnection is out of interest, hence using the general term NoC.

2.2. Safety and Certification

The ultimate goal in civil aviation is the safe transportation of passengers. An aircrafts ability *"to be operated in flight and on the ground without significant hazard to aircrew, ground crew, passengers or to third parties"* [MMA (2010)] is defined as airworthiness. In other words, airworthiness describes whether an aircraft is worthy of safe flight. Safety, in turn, is defined as a system property, that ensures the *"absence of catastrophic consequences on the user(s) and the environment"* [Laprie et al. (2004)]. Ahead of the first flight of an airplane, both, airworthiness and safety have to be verified by legal authorities. If the manufacturer can prove the adherence to the authority's regulations, the aircraft gets the certification for safe flight, permitting operational use. The certification authorities for Europe and the United States of America are the European Aviation Safety Agency (EASA) and the Federal Aviation Administration (FAA). Their respective regulations are called Joint Aviation Regulationss (JARs) and Federal Aviation Regulationss (FARs), forming the legal bodies for certification. Based on the regulations industry standards have been developed, e.g. for the system development process [SAE (2010)], the hardware life-cycle process [RTCA (2000)] and the software life-cycle process [RTCA (2012)], cf. Figure 2.1 for their respective relations. As long as the standards are accepted by the authority, it is sufficient to prove adherence to the standards to achieve certification. The processes defined within these standards are highly driven by requirements, traceability and documentation to seamlessly cover the certification requirements.

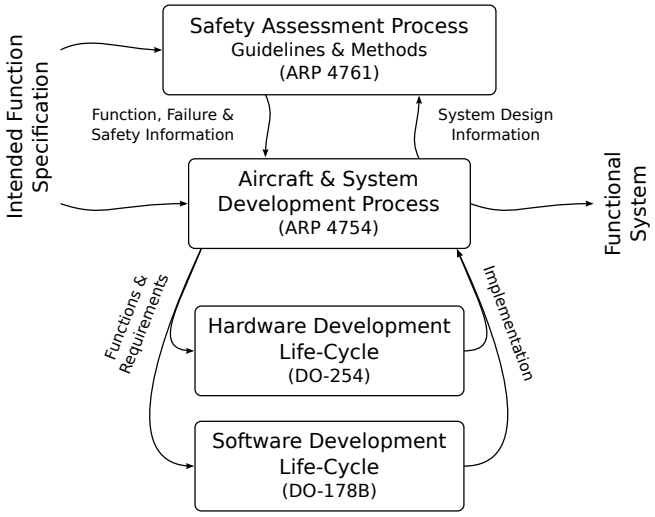


Figure 2.1.: *Relations between avionics guideline documents, cf. [SAE (2010)].*

The safety of airplanes is carefully assessed through accident statistics, for instance annual safety reviews as in [EASA (2012)]. They include information such as the absolute numbers of occurrences, location, severity and incident causes. Based on such statistics different safety levels are defined, cf. Table I. Depending on the actual standard they are called differently, for instance DO-178 [RTCA (2012)] uses software levels whereas DO-254 [RTCA (2000)] defines system development assurance and hardware design assurance levels. Similar definitions of assurance levels are used in other industry domains, e.g. the IEC 61508 [IEC (2010)] defines Safety Integrity Level (SIL), and automotive industry uses Automotive Safety Integrity Level (ASIL), cf. [ISO (2011)]. Throughout this thesis the term Design Assurance Level (DAL) is used to refer to safety levels.

DALs are assigned per application, based on their failure probability, the impact of its failure on higher level applications and implemented mitigation mechanisms to handle possible failures.

As an example, the passenger notification system is from high criticality, since it is, for instance, required to notify passengers in case of a fire. But, a megaphone as mitigation could decrease the actual assurance level of the involved applications. Considering this, assurance levels are derived from accident statistics as [EASA (2012)] and empirically based on pilot experience and previous aircrafts. Table I summarizes safety levels as they are defined in [SAE (2010), RTCA (2012), RTCA (2000), SAE (1996)].

Table I.: *Safety level definitions according to [SAE (2010), RTCA (2012), RTCA (2000), SAE (1996)].*

DAL [RTCA (2012)], [RTCA (2000)]	A	B	C	D	E
Severity [SAE (2010)]	Catastrophic	Hazardous	Major	Minor	No Effect
Probability [SAE (1996)]	$< 10^{-9}$	$< 10^{-7}$	$< 10^{-5}$	< 1	-
Effect on Passenger	Multiple Fatalities	Serious of Fatal Injury to Small Number of Persons	Physical Distress, Possibly Including Injuries	Physical Discomfort	None
Application Example	Breaks, Steering, Navigation	Smoke Detection	Passenger Notification	Cabin Illumination	Floor Heater

Naturally, the higher the assurance level, the higher the effort that has to be put in design, implementation, verification and testing. For instance, DO-178C [RTCA (2012)], concerned with the software in an aircraft, defines the following code coverage metrics for software verification:

Requirements Coverage: Every high- and low-level software requirement is implemented. Whereas, high-level requirements are derived from *"system requirements, safety-related requirements, and system architecture"* [RTCA (2012)]. Low-level requirements are derived from *"high-level requirements, derived requirements, and design constraints from which Source Code can be directly implemented without further information"* [RTCA (2012)].

Statement Coverage (SC): *"Every statement in the program has been invoked at least once"* [RTCA (2012)].

Decision Coverage (DC): *"Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once"* [RTCA (2012)].

Modified Condition/Decision Coverage (MC/DC): *"Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome"* [RTCA (2012)]. [Hayhurst et al. (2001)] provides a detailed description of different coverage metrics, including practical examples on MC/DC.

Considering the verification effort, MC/DC poses the most stringent requirements. Accordingly, Table II shows the relation between an applications' DAL, the required software coverage, and the

considered cost impact.

Table II.: *Relation between DAL and required software coverage metrics as defined by [RTCA (2012)] Table A-7, and associated costs, cf. [HighRely (2009)].*

DAL	A	B	C	D	E
Coverage	DAL B + MC/DC	DAL C + DC	DAL D + SC	Requirements Coverage	-
Costs	DAL B + 5%	DAL C + 15%	DAL D + 30%	DAL E + 5%	-

As can be seen in Table II, the higher the assurance level, the higher the coverage requirements and associated costs. Besides increased costs, higher coverage requirements also entail different assumptions on the behaviour of applications. For example, the probability for DAL-A applications to behave as expected is much higher than for DAL-D applications. Hence, more intensive runtime control mechanisms might be required for the latter.

In essence, this section briefly described the relation between safe operation of an airplane and certification requirements. Also the categorisation into DAL and the different associated efforts and assumptions have been presented. It is crucial to understand the requirements driven nature of all processes. It is also important to remember, that every part of an aircraft system has to pass certification. Respectively, certification poses different requirements the development and verification of applications, depending on their DAL. In the context of this thesis timing requirements and analysability of applications are from special concern.

2.3. The Partitioning Concept

2.3.1. Avionics Architecture Evolution

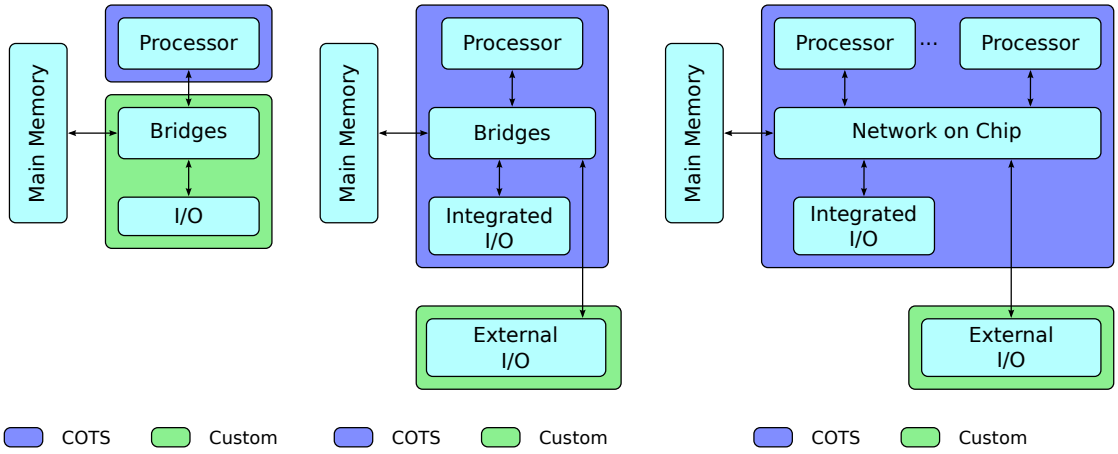
The goals and requirements of partitioning are best explained by the evolution of avionics architectures over recent decades, changing from fully, physically separated systems towards integrated platforms.

In the past, avionic systems have been implemented following a *federated* design [Watkins and Walter (2007)]. Each avionics function, e.g. flight control or cabin pressure management, has been implemented on self-contained modules referred to as Line Replaceable Modules (LRMs). Each LRM contained a processor unit, memory and peripheral interfaces, cf. Figure 2.2(a). Likewise, separate power supplies were required. The design often consisted of a COTS host processor and custom designed bridges and I/O interfaces. Beneath others, custom bridges allowed the manufacturer to detect and handle processor malfunctions and mitigate errate. This enabled the use of commercial processors even if their design and manufacturing processes did not match certification requirements.

Today's avionics systems are often designed according to the Integrated Modular Avionics (IMA) concept, which is defined in [RTCA (2005)]. It has been introduced in order to reduce the Space, Weight and Power (SWaP) requirements of the avionics system and the fuel consumption of the airplane [Wilson and Preyssler (2008), Prisaznuk (1992)]. Instead of dedicated LRMs per application, IMA architectures implement certain basic components that are shared between applications, e.g. power supply, core processor or standard I/O modules [RTCA (2005), Prisaznuk (1992)]. The IMA concept also enabled mixed-criticality systems, which integrated multiple functions of different DAL on the same LRM. In consequence, the tighter integration and potential interferences between applications of different DAL poses additional isolation requirements. To ensure similar

separation than between functions on a federated architecture the *partitioning* concept, discussed later in this section, has been introduced. In conjunction with the increased system integration also more integrated SoC processor elements are deployed as main computing units. In contrast to host processors, SoCs already include bridges, memory controller and common I/O interfaces, cf. Figure 2.2(b). Hence, it is no longer possible to mitigate malfunctions of the processor within a custom bridge. In consequence SoC-based systems pose additional challenges for certification [EASA (2011), Kinnan (2009)].

Following the trend towards increased integration the next evolutionary step are multi-core processors and MPSoCs as shown in Figure 2.2(c). In addition to integrated bridges and I/O interfaces, MPSoCs consist of multiple processing cores and some kind of interconnecting NoC. As discussed in Chapter 1, this further complicates separation of applications due to non-deterministic access latencies and unknown NoC design.



(a) Federated architecture (past)

(b) IMA SoC-based architecture (present)

(c) IMA MPSoC-based architecture (future)

Figure 2.2.: *Processing unit design for avionics systems throughout history.*

Considering the architectural evolution depicted in Figures 2.2(a) to 2.2(c), it can be seen that increasingly more COTS components are utilised. Even though this can reduce development costs and increase performance, it also withdraws the possibility to mitigate malfunctions in relatively simple, custom designed elements, developed according to certification requirements.

2.3.2. Partitioning in Avionics Systems

As mentioned, the partitioning concept has been introduced in conjunction with the IMA architecture. Its purpose is to ensure the same level of separation between applications on an IMA system as on former federated platforms in order to avoid any unindented influences. Equivalently, Rushby defined the *gold standard for partitioning* as follows: *"A partitioned system should provide fault containment equivalent to an idealized system in which each partition is allocated an independent processor and associated peripherals and all inter-partition communications are carried on dedicated lines"* [Rushby (2000)].

The following description is based on the ARINC 653 [ARINC (2003)] avionics standard. It defines the interface between applications and the operating system layer in an IMA system. Similar concepts can be found in the AUTOSAR standard for the automotive industry [AUTOSAR (2013)].

Partitioning is a concept for spatial and temporal isolation of applications. Applications are abstracted to *partitions*. By definition partitions are functionally separated entities such, that no partition can influence another partition in spatial or temporal domain. Id est, a partition cannot access another partitions' memory space nor influence its timing. This also includes fault containment, i.e. a fault of one partition does not cross partition boundaries. A so-called *Separation Kernel (SK)* is used to enforce partitioning at runtime. According to [Rushby (1981)] *"the task of a separation kernel is to create an environment which is indistinguishable from that provided by a physically distributed system"*. Even though Rushby defined SK for security purposes the definition similarly applies to safety-critical systems.

Beneath the functional aspects of partitioning, it is also required to enable incremental development and certification. This is especially required if partitions are developed by different suppliers. Figure 2.3 shows an abstracted view of a partitioned system. As described, the SK abstracts the hardware and provides isolated environments for each partition.

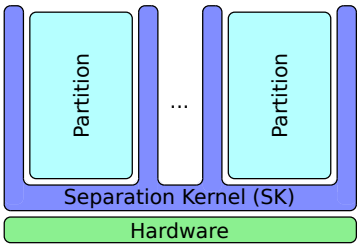


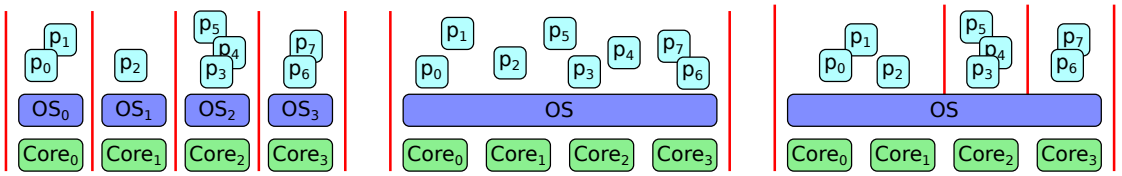
Figure 2.3.: *The partitioning concept.*

Spatial Partitioning

Spatial partitioning covers the separation of address spaces and the assignment of peripheral devices to partitions [Rushby (2000)]. Although address space separation can be implemented in software, it is commonly based on hardware acceleration by Memory Protection Units (MPUs) or MMUs. The functionality of an MPU allows the definition of memory spaces that a partition is allowed to access. Each access is checked for permission, once an access is outside the defined range an exception is raced, signaling the violation to the SK. Additionally, MMUs allow address translation from virtual to physical addresses. This enables the use of the same virtual addresses by multiple partitions. The SK is responsible to configure MPU/MMU and handle the related exceptions. The mapping from partitions to address spaces can either be defined statically or determined at runtime. Similar functionality is provided for I/O devices by the means of I/OMMUs. They prevent Direct Memory Access (DMA)-capable peripherals from accessing addresses outside a defined range, e.g. to avoid one partition to indirectly access another's partition memory by using DMA transfers. The assignment from I/O devices to partitions is generally handled via devices drivers, either according to a static system configuration or on demand in dynamic systems. To avoid the overhead of driver abstractions, self-virtualising devices have been introduced, cf. [PCI-SIG (2007), Liu (2010)]. Beneath others, this allows the direct access of partitions to a subset of the registers and memory of I/O devices. Obviously, it has to be ensured, that the implementation does not allow a partition to violate the system safety by directly controlling I/O devices,

Transitioning from single-core to multi-core processors does not pose conceptually new challenges to spatial separation. Nevertheless, additional design questions arise. The most dominant question is the assignment between partitions and processing cores. [Tanenbaum (2007)], beneath others, defines two assignment schemes. The first scheme allocates separate memory spaces for

each core with a private copy of the operating system. Although Tanenbaum did not term it specifically, but today this is commonly known as Asymmetric Multiprocessing (AMP) system. For the second scheme, names Symmetric Multiprocessing (SMP), all cores share a single operating system copy. Transferring those definitions to partitioned environments leads to the following: In AMP systems each partition is assigned to a single core, with a separate SK instance controlling it, cf. Figure 2.4(a). Accordingly, SMP systems allow any partition to be executed on any core, while all cores are controlled by a single SK instance, cf. Figure 2.4(b). In addition to AMP and SMP some operating system suppliers define a third scheme called Bound Multiprocessing (BMP). According to [QNX (2013)], BMP allows to specify a processor core affinity, to define a range of cores that a partition is allowed to be scheduled on, cf. Figure 2.4(c). Since BMP is only a special case of SMP it is not further considered. The scheduling boundaries in Figures 2.4 illustrate the assignment between partitions and cores.



(a) AMP (b) SMP (c) BMP

Figure 2.4.: *Comparison of AMP, SMP and BMP processor core assignment schemes. Scheduling boundaries illustrate the assignment between partitions p_i, SK instances and cores.*

Further research areas, beneath processor core assignment schemes, are for instance different MPU and MMU implementations, cf. [Hattendorf et al. (2012)], and virtualisation techniques for peripheral devices [Willmann et al. (2007), Liu (2010), Münch et al. (2014)].

Throughout this thesis spatial partitioning is assumed to be sufficiently solved, since it is not solely required by safety-critical systems.

Temporal Partitioning

The mechanism of temporal partitioning ensures the decoupling of partitions' timing behaviour, i.e. the execution of one partition does not influence the timing of any other partition [Rushby (2000)]. For instance, a deadline violation of one partition does not have any effect on other partitions.

In single-core systems, temporal partitioning is implemented via time multiplexing, where each partition gets a certain interval for execution. During that time no other partition is active. The activation and suspension of partitions is controlled by a scheduling algorithm. Depending on the nature of the system, either online/dynamic or offline/static scheduling techniques can be used. In general online scheduling can react more flexibly on (dynamic) events, whereas the sequence and time of execution is pre-determined for offline scheduling. Consequently, online scheduling requires more runtime overhead in favor of flexibility. Popular scheduling algorithms for real-time systems are Rate-Monotonic Scheduling (RMS) and Earliest Deadline First (EDF), cf. [Liu and Layland (1973)]

The ARINC 653 standard defines a cyclic scheduling scheme with statically defined time windows for each partition. A partition is characterized with its period, duration and offset relative to the start of the scheduling cycle. Consequently, the deadline of a partition instance is constituted

by its relative activation point and duration. The period of a partition depends on the system functionality, its duration needs to be bounded via timing analysis and the offset into the scheduling cycle is calculated during system integration. The length of the scheduling cycle is called major time frame. It is defined as *"a multiple of the least common multiple of all partition periods"* [ARINC (2003)]. The sequence of partitions within the major time frame is defined statically. At runtime this sequence is repeated cyclically. Temporal partitioning is ensured by suspending partitions that reach their deadline. For illustration, Table III defines four partitions with their period, execution time and offset to the beginning of the major time frame. Figure 2.5 shows the respective sequence of partitions.

Table III.: *ARINC 653 exemplary configuration for four partitions.*

Partition	Offset	Duration	Period
p_0	0	10	40
p_1	10	20	80
p_2	30	10	80
p_3	130	30	160
	Major Time Frame		160

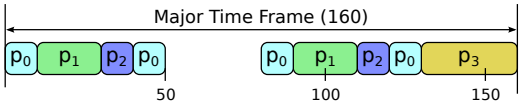


Figure 2.5.: *Example for ARINC 653 major time frame with four partitions, according to the configuration shown in Table III.*

Besides the execution of the partitions, major issues for temporal partitioning in single-core processors are interrupt handling and the use of DMA transfers. The issues can be summarized as follows, a more detailed description can be found in [Littlefield-Lawwill and Kinnan (2008)].

Interrupts are external events that redirect the processor control flow to the Interrupt Service Routine (ISR). Hence, the time used for interrupt handling is taken from time window of the active partition. This is no issue as long as this is considered during timing analysis. But, temporal partitioning is violated once interrupts, dedicated to another than the active partition occur. Hence, usually non-deterministic interrupts are disabled in IMA systems. Only periodic interrupts, such as timers, are used. Unfortunately, interrupts are often required for efficient device interaction. Otherwise polling implementations need to be used, which are likely to reduce the achieved bandwidth due to additional software interaction.

Besides interrupts, the use of DMA transfers or DMA-capable devices can violate temporal partitioning. DMA engines can issue memory accesses independently from the processor, possibly causing collisions on, or fully block the memory interconnect. This is, similar to interrupts, no issue for temporal partitioning as long as the DMA transfers belong to the active partition. A partition violation exists, once a DMA operation is in progress that has been triggered by another than the active partition. Hence, it has to be ensured that all issued DMA transfers can be finished within the time window of the issuing partition. Corresponding implementations are for instance based on adequate driver abstractions or dedicated I/O partitions.

As already discussed in Chapter 1.1, multi-core processors introduce true parallel execution via multiple processing cores. With respect to temporal partitioning they cause similar problems than DMA-capable devices. Whereas, they can not be handled in device drivers or separate I/O partitions. Regarding incremental development, it is most likely that partitions running in parallel

are developed by different suppliers, hence the introduced interferences are generally unknown. As long as such interferences are not fully avoided by complete separation of partitions, e.g. through serialisation of shared resource requests, the need to be accounted differently. For that reasons, the definition of temporal partitioning is relaxed within this thesis in a sense, that temporal partitioning does not provide full isolation of the temporal behaviour of partitions, but instead guarantees a bounded influence.

In summary, partitioning is an essential property of avionics systems. As discussed spatial separation is considered to be solved for single- and multi-core processors. Temporal partitioning on the other hand requires additional effort in order to solve the question of unknown shared resource interferences between processing cores and I/O devices. Hence, an approach for timing analysis, enabling temporal isolation on multi-core processors, is proposed within this thesis.

2.4. Worst-Case Execution Time Analysis

Timing analysis is essential for the design of real-time systems. Especially the computation of WCET bounds is obligatory to schedule applications such, that all of them meet their deadline. The resulting execution time bounds are required to safely upper bound the applications WCET. Furthermore, the analysis results should be as close to the real WCET as possible. This is also referred to as tightness of the analysis. Figure 2.6 depicts the basic terminology for timing analysis related to potential program executions. The black graph shows all theoretically possible executions, while the white graphs represent the observable executions. The minimal and maximal Observed Execution Times (OETs) limit the measured executions. Correspondingly, the Best-Case Execution Time (BCET) and WCET bound the possible application behaviour to the lower and upper end. The bounds computed during timing analysis are shown as lower and upper timing bounds. Naturally, the lower bound is smaller than the BCET and the upper bound is higher than the WCET, since they are both required to safely bound the best-case and worst-case behaviour, respectively. The interval between lower and upper timing bound is named *timing predictability* of an application or computing architecture. It quantifies the predictability of a given computing architecture or application. The *tightness* of an analysis is defined as the difference between the computed timing bound and the BCET/WCET, respectively. The smaller the difference, the tighter the timing bound, the higher the difference, the larger the underestimation for best-case and overestimation for worst-case analysis. An inherent problem of timing analysis is to quantify the tightness of an analysis, even though tightness is clearly defined. According to the halting problem [Turing (1936), Davis (1958)], it is impossible to formulate an algorithm, that can decide whether a program terminates for arbitrary input data. Likewise, it is not possible to calculate its execution time. Consequently, BCET and WCET are theoretical, but in general unknown bounds. To, nevertheless quantify the tightness of a timing analysis it is common practice to compare observed and analysed timing bounds [Souyris et al. (2005), Thesing et al. (2003)]. Since this thesis targets worst-case timing analysis only upper timing bounds are discussed in the remainder.

First approaches in WCET analysis have been presented in the late 1980s [Shaw (1989), Puschner and Koza (1989)]. The relatively simple processor architectures allowed to determine the execution time of single instructions via the processor manual. Today's processor architectures are considerably more complex, including multi-stage pipelines, branch predictors and multi-level memory hierarchies. In effect the timing of an individual instruction does no longer solely depend on that instruction. Rather, it depends on previously executed instructions and their changes to the pro-

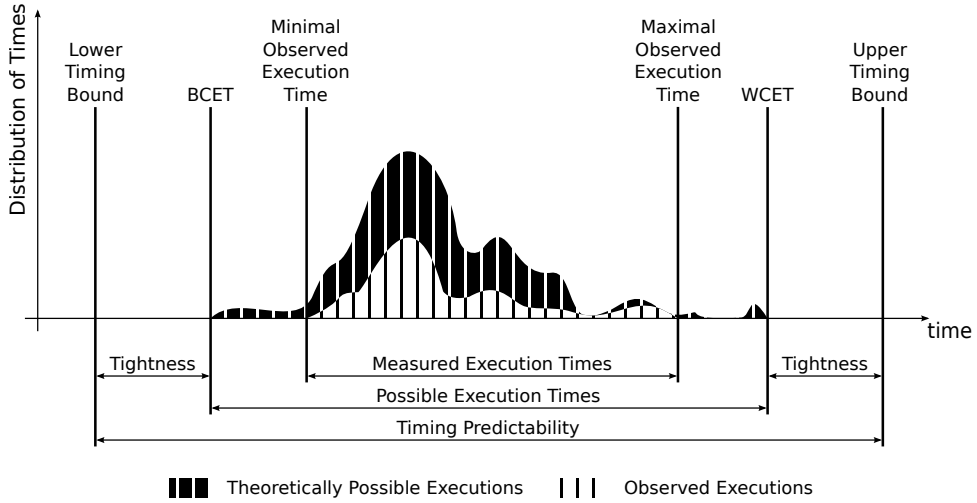


Figure 2.6.: *Timing analysis terminology, cf. [Wilhelm et al. (2008)].*

cessor's internal state. Hence, in modern architectures, a single instruction cannot be analysed in isolation, instead its mutual dependencies with other instructions have to be considered. This is also know as execution history dependence [Wilhelm et al. (2008)]. Without disrespecting the variety of individual approaches, three approaches are commonly applied nowadays: end-to-end measurement, static analysis and hybrid measurement-based analysis. For end-to-end measurements the input parameters of an application are chosen such, that worst-case behaviour is triggered. The execution time is measured for the whole application or for specifically interesting functions, deriving a distribution of observed execution times, as shown in Figure 2.6. In contrast static and hybrid approaches are based on detailed analysis of application and target architecture. Corresponding implementations realize a more or less standard architecture for timing analysis as described in details in the following and referred to throughout this thesis.

2.4.1. Architecture for Timing Analysis

Modern timing analysis accommodate the fact that the execution time of an application depends on the application, its input data and the target computing platform. Hence the analysis is split into phases to analyse the program flow, the characteristics and implementation details of the target architecture and the final computation of the execution time bound. Over time the quasi standard architecture, as shown in Figure 2.7, has evolved.

Control Flow Construction and Analysis

The control flow analysis consists of two steps, the control flow construction and control flow analysis. They are based on the application binary and/or source code representation.

During the control flow construction a control flow representation, e.g. Control Flow Graph (CFG) or Abstract Syntax Tree (AST), of the application is generated. This representation contains a superset of all possible execution paths. In the following only CFGs are considered. A CFG is a directed graph $G = (V, E)$. With V being the set of all nodes within G, where each node represents a basic block. A basic block is defined as a sequential block of instructions with only one entrance and exit point [Allen (1970)]. The edges E in G indicate control flows from the source to the target basic block [Allen (1970)]. Often, two special basic blocks are used as entry

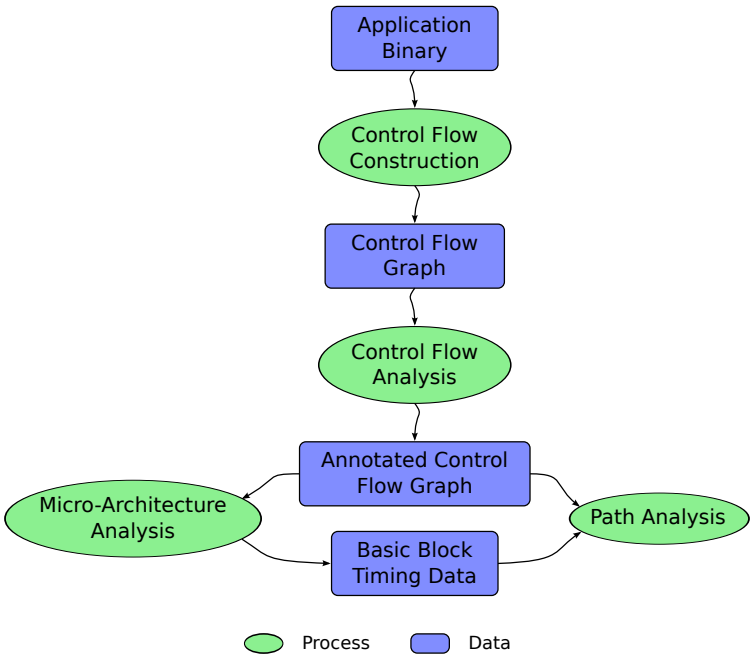


Figure 2.7.: *Quasi standard architecture for timing analysis.*

and exit nodes in the CFG. A path through the CFG is a sequence of nodes and edges from the entry to the exit node. Subsets of the CFG, which do not start at the entry or end at the exit node are called sub-paths. Listing 2.1 and Figure 2.8 depict an exemplary source code snipped and the respective CFG.

```
1  // basic block 0 lines [2, 4]
2  int x;
3
4  if(a == b)
5      // basic block 1 line 6
6      x = 2;
7  else
8      // basic block 2 line 9
9      x = 3;
10
11 // basic block 3 lines [12, 13]
12 do{
13     if(a > b)
14         // basic block 4 line 15
15         i--;
16     else
17         // basic block 5 line 18
18         i++;
19
20     // basic block 6 lines [21, 22]
21     x--;
22 }while(x > 0);
```

Listing 2.1: *CFG example - code snippet.*

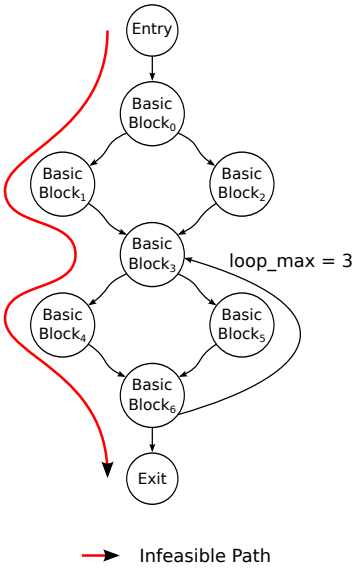


Figure 2.8.: *CFG corresponding to Listing 2.1.*

During control flow analysis the program control and data flow is analysed to identify so-called flow facts [Engblom and Ermedahl (2000), Stappert et al. (2001)], as well as infeasible paths [Park (1993), Suhendra et al. (2006)]. Flow facts, for instance contain execution frequencies of paths

and sub-paths, loop bounds, depths of recursive function calls, and relations between different paths and sub-paths. The information acquired during control flow analysis are annotated to the CFG. Figure 2.8 shows an annotated loop bound and an infeasible path due to contradicting conditions for basic block$_1$ and basic block$_4$, i.e. if basic block$_1$ is executed basic block$_4$ will never be executed, hence the path *entry* $\rightarrow$ basic block$_0$ $\rightarrow$ basic block$_1$ $\rightarrow$ basic block$_3$ $\rightarrow$ basic block$_4$ $\rightarrow$ basic block$_6$ $\rightarrow$ *exit* will never occur during execution.

This first phase of timing analysis is similar for static and hybrid approaches.

Micro-Architecture Analysis

The micro-architecture analysis is performed to annotate upper bounds for execution the time of each basic block within the CFG. As mentioned above, previous, relatively simple processor architectures could be analysed solely with the instruction execution times given in the processor manual. Hence, the timing of different basic blocks could be calculated independently and the execution time for a sequence of basic blocks is constituted by the sum over the respective basic blocks. Hence, the execution time bound ($t_{0,1}$) for the sub-path basic block$_0$ $\rightarrow$ basic block$_1$ has been calculated as $t_{0,1} = t_0 + t_1$. Unfortunately, this is no longer a valid approach for today's processor architectures. Due to the instruction history dependence, the execution time of the paths basic block$_0$ $\rightarrow$ basic block$_1$ and basic block$_1$ $\rightarrow$ basic block$_0$ could be very different, whereas they would have been equal for simple architectures. Nowadays, history dependence is covered by context sensitive computation of execution times. The context is an abstract processor state containing all relevant architecture features. Whether a feature is relevant or not depends on the level of abstraction that is used for analysis. Context changes are analysed for each basic block, i.e. in the above example basic block$_0$ $\rightarrow$ basic block$_1$, the execution time for basic block$_1$ is analysed in the context established by basic block$_0$. In consequence, each basic block gets an execution annotation for each context in which it might be executed.

The micro-architecture analysis is performed differently by static and hybrid approaches. Static analysis techniques are based on a model of the target hardware, while hybrid approaches use execution times measured on the target hardware.

The model for static analysis is more or less complex, depending on the applied abstraction. It directly influences the tightness as well as the complexity of the architecture analysis. For instance, if caches are modeled each access is classified either as cache hit, miss or unknown, which might also differ depending on the context they are executed in. This is obviously more complex than assuming every access to be a cache miss. On the other hand, considering the execution time, such abstraction can result in huge deviations between computed bound and real WCET since cache and memory access times differ in order of magnitudes. Beneath the hardware also the input data are abstracted in order to cover a superset of all possible execution paths. If the abstracted model is correct static timing analysis computes safe upper bounds for application's WCET.

On the other hand, hybrid approaches have the advantage of the real hardware, eliminating the necessity for abstraction. The execution time of basic blocks or sequences of basic blocks are measured while the input data are chosen to trigger worst-case behaviour. History dependence is accounted through triggering the execution of basic blocks under various contexts by altering the input data, forcing specific execution paths. The main issue for this approach is to gain sufficient coverage in terms of input data and measured paths through the application. Considering the variability of input data and complexity of applications it is not possible to gain full coverage, i.e. the computed execution time bounds cannot be proven to safely upper bound the WCET.

Beneath static and hybrid analyses a new technique emerged in the last years, called probabilistic timing analysis [Burns and Edgar (2001), Bernat et al. (2002), Bernat et al. (2003)]. Instead of

fixed execution time bounds per basic block, execution time distributions are annotated. Therefor, random arbitration and replacement schemes are assumed, instead of the traditional strategies. For instance, instead of Least Recently Used (LRU) cache replacement it is assumed, that each cache line is selected with the same probability for eviction every time. This is required in order to be able to assume equal distribution and independence of events [Petters (2002)]. A comparison of random and LRU cache replacement strategies revealed, that the LRU provides higher performance than random, even though requiring increasing computing complexity [Gallo et al. (2012)]. As commercial hardware is generally targeted towards average-case performance, it is questionable whether random arbitration and replacement schemes will be implemented in COTS devices. As the focus of this thesis is on COTS components probabilistic timing analysis is not further considered.

Path Analysis

The final phase of timing analysis is the path analysis. Based on the CFG, the annotated flow facts and the basic block timings a bound for the WCET and one or more corresponding paths are computed. In literature different approaches have been proposed, among others tree-based [Puschner and Koza (1989), Chapman (1994), Lim et al. (1995), Colin et al. (2002), Betts (2010)], path-based [Healy and Whalley (1999), Stappert et al. (2001)] and Implicit Path Enumeration Technique (IPET)-based [Engblom and Ermedahl (2000), Puschner and Schedl (1995), Li and Malik (1995), Ferdinand et al. (1997)] algorithms.

Tree-based approaches rely on ASTs as control flow representation. A corresponding timing schema defines how the execution time of a sequence of nodes is calculated, depending on the relation between them. For instance if two nodes constitute the if and else branch of an alternative, the resulting execution time is the maximum over their execution times. Although tree-based approaches are relatively simple, they cannot consider data acquired during control flow analysis [Betts (2010)] (p. 133).

Path-based algorithms explicitly traverse through the CFG to find the longest path. Whereas IPET algorithms either formulate the problem as an Integer Linear Program (ILP) or Constrained Program (CP) accounting for the flow facts.

2.4.2. Overestimation of Execution Times

As discussed, one of the goals of timing analysis is to safely upper bound the WCET. By design only static timing analysis can be proven to fulfil this requirement. Nevertheless, depending on how execution times are measured, also hybrid approach might overestimate the WCET, i.e. the computed upper timing bound is larger than the actual WCET. For example, if the execution times of the basic blocks that contribute to the longest path are measured with different input data and contexts the resulting bound is calculated based on timings that might never be observed in this particular combination during real execution. As explained, this cannot be proven for the general case.

Independent of the individual reason, overestimation of WCETs is a major concern for the efficiency of real-time embedded systems. Since computed bounds are used during scheduling to assign processor time to the applications assigned but practically unused time results in idle cycles and reduced utilisation. In order to increase the tightness of an analysis it is important to understand the possible sources of overestimation, summarized as follows:

Unknown Hardware and Application Context: They mainly arise due to the complexity of computing architectures and applications. For instance, application code complicates analysis by

using computed branches and indirect addressing, whose final targets cannot be resolved during analysis. Also input parameter dependencies cannot be resolved without manual annotation. This can, for example lead to analysed program paths, that are not feasible during real execution. Since static timing analyses uses a model of the target architecture, history-based features, such as caches or branch prediction are additional sources that introduce large context search spaces. Refer to [Wilhelm et al. (2008), Wilhelm et al. (2009), Cullmann et al. (2010)] for further details.

Analyses Inadequacy and Abstraction: For reasons of complexity the analysis needs to abstract the behaviour of application and architecture [Wilhelm et al. (2008)]. Depending on the accuracy of those abstractions some details, e.g. border cases that would speedup execution, might not be modeled. Hence the resulting timing bound will overestimate the WCET. Furthermore, the different contexts created during analysis need to be abstracted, allowing the trade-off between analysis time and accuracy.

Undetailed Documentation: Correct and complete documentation is an essential requirement to create an architecture model that adequately represents the target platform. Unfortunately, arbitration and replacement policies are not necessarily documented in full details, e.g. such that optimisations are not fully described or that documentation differs from the actual silicon. For instance, Atanassov et al. [Atanassov et al. (2001)] observed differences between the timing formulae reported in the manual and those of the real hardware for an Infineon C167 processor.

2.4.3. Timing Composability and Anomalies

"A system is said to be composable with respect to a specific property if the system integration will not invalidate this property once the property has been established at the subsystem level" [Kopetz (1997)]. In timing analysis composability is required once divide-and-conquer approaches are used, e.g. if different the micro-architectural analysis and path analysis are performed separately to reduce analysis complexity [Theiling and Ferdinand (1998), Theiling et al. (2000)]. After the individual analyses have been performed the results need to be combined to calculate the final WCET bound. This combination of results is done via a composition operation. Therefore, the compositionality of the sub-results is inherently assumed. Considering separate cache and pipeline analyses for a basic block this applies as follows: During pipeline analysis it is determined which instructions utilise which pipeline stages and cache analysis classifies memory access as cache hit, miss or unknown. Afterwards the cache information are used to determine the memory access latencies, while the required cycles through the pipeline are calculated based on the pipeline information. The combination of both analyses constitutes the WCET bound of the basic block. Composability is further required when combining bounds of different basic blocks to calculate the timing bound of an execution path.

Unfortunately, so-called timing anomalies violate the assumption of timing composability. Informally described, timing anomalies are counter-intuitive timing behaviours, where the local worst-case does not constitute the global worst-case, e.g. a cache miss, generally the local worst-case, leading to shorter execution time than a cache hit. In effect, both cases need to be considered during analysis, which drastically increases the search space and hence the complexity of the analysis. Figure 2.9 shows an exemplary case with 4 instructions, A to D, that are scheduled on two processor resources, r_0 and r_1. Each diagram shows the number cycles that an instruction occupies a resource. Arrows below the x-axis indicate the cycle at which an instruction is ready for execution. The respective constraints are summarized in Table IV. Arrows between instructions illustrate data dependencies. A corresponding code sequence is shown in Listing 2.2. Figure 2.9(a) depicts the timing behaviour in the case of a cache hit. Figure 2.9(b) depicts the cache miss

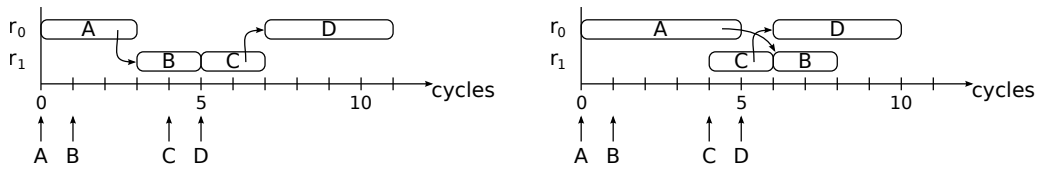
case, with instruction A being extended by 2 cycles due to the miss. As can be seen, the overall execution of the sequence is decreased by 1 cycle in case of the cache miss instead of being delayed.

```
lwz     r1, 0x0(r2)     // A
addi    r1, r1, 0x4     // B
sub     r3, r4, 0x1     // C
stw     r3, 0x40(r31)   // D
```

Listing 2.2: *Cache hit/mis anomaly - code snippet.*

Table IV.: *Cache hit/miss anomaly - constraints.*

Instruction	Resource	Ready Cycle	Depends On
A	r_0	0	-
B	r_1	1	A
C	r_0	4	-
D	r_1	5	C



(a) cache hit (b) cache miss

Figure 2.9.: *Cache hit/miss anomaly - timing behaviour, cf. [Wenzel et al. (2005)].*

Over the years timing anomalies have been studied in several publications. They have first been mentioned in the context of timing analysis by [Lundqvist and Stenström (1999)] and later by [Engblom and Jonsson (2002)]. The authors of both publications state, that timing anomalies can only appear if out-of-order resources are utilised. Hence in-order architectures are considered to be anomaly free. This assumption is proven wrong by [Wenzel et al. (2005)], who present examples for timing anomalies in architectures that solely consist of in-order resources. They identify in-order resources with overlapping, but not equal, abilities to also suffer timing anomalies. The reason being, are resource allocation decisions that result in a different resource utilisation for the same sequence of instructions, if the latency of one of those instructions is varied. The authors formulate this as the *resource allocation criterion*. The final conclusion states, that timing anomalies are absent in architectures which do not allow resource allocation decisions. Based on the rather informal definitions of previous work, the [Reineke et al. (2006)] present a formal definition of timing anomalies: *"A hardware model exhibits timing anomalies, if there exists a program P with a finite sequence σ through P, and a non-local worst-case path $\pi \in \Pi$ such, that $|\pi| > |\pi'|$ for all local worst-case paths $\pi' \in \Pi$"* with Π being the set of all micro-architectural paths corresponding to the sequence of instructions σ through P. In cases where the effect of a timing anomaly cannot be bounded, it is called a domino effect [Lundqvist and Stenström (1999)].

Based on the presence of timing anomalies [Wilhelm et al. (2009)] propose a classification of computing architectures and their respective timing models as follows:

Fully Timing Compositional Architecture: This architectures/models do not suffer timing anomalies. Hence timing analysis can rely on local worst-case decisions to produce safe timing bounds. For instance, if a cache access is not safely classified as hit or miss, it is safe to only follow the path that results from a cache miss.

Compositional Architecture: This kind of architectures/models exhibit timing anomalies but no domino effects. In general, timing analysis has to consider all possible execution paths but some trade-offs between precision and analysis complexity are possible. Considering the undecided

cache access, generally both, the hit and the miss path have to be considered. But, since the potential effect of a timing anomaly is bounded, it is sufficient to only follow the local worst-case path, while adding a constant factor for the potential timing effect of the non-local worst-case path. This reduces analysis complexity but for the cost of precision since a constant penalty for the potential timing effect is added.

Non-compositional Architecture: Non-compositional architectures/models suffer bounded anomalies as well as domino effects. Timing analysis always has to consider all possible execution paths to produce correct results. In case of the undecided cache accesses, it is required to follow the cache hit as well as the cache miss path.

In summary, timing anomalies are an issue of complexity of processors and non-determinism in the applied architecture timing model, while the underlying hardware is deterministic. They lead to deviations between the timing model and the actual hardware such, that predicted execution time bounds become wrong [Wenzel et al. (2005)]. According to [Reineke et al. (2006)] non-determinism is introduced by abstraction, which is necessary to deal with the complexity of the real hardware. Hence the consideration of timing anomalies in timing analysis greatly increases complexity. Timing anomalies need to be considered for out-of-order, as well as for in-order processors, i.e. in fact for all modern processors. In the context of this thesis it is important to note, that timing anomalies can be handle for instance through serial-execution and code modifications as discussed in [Lundqvist and Stenström (1999)]. Furthermore, it can be analysed if dynamic resource allocations happen for the given combination of target hardware and analysed software [Wenzel et al. (2005)]. Consequently, timing analysis approaches that implement separate analyses of resources, such as the architecture described above, can be applied.

2.4.4. Timing Analysis and Design Assurance

The context of safety-critical applications and especially the defined DALs enable the use of diverse timing analysis methods for applications of different assurance levels. The following is a short review of the different timing analysis methods, their advantages, disadvantages, and applicability for respective DALs.

End-to-end measurements only use variable input data to trigger worst-case behaviour. Considering the complexity of applications and the potential range of input data, it is intuitively understood that it requires intensive testing in order to prove sufficiently low failure probabilities. Furthermore, it has to be ensured that worst-case behaviour in the underlying hardware is triggered. For instance, the authors in [Butler and Finelli (1993)] report a testing effort of 10^6h ($114a$) to prove a failure probability of 10^{-9} for a $10h$ operation, using 10000 replicated testing systems. Accordingly, a DAL-A software (failure probability of 10^{-9} per flight hour) would require 10^5h ($11a$) of testing under equal conditions

Hybrid approaches, such as [Betts (2010), Rapita (2014)], are based on the architecture described above. This enables more detailed analysis than pure end-to-end measurements. Also additional effort is spend to increase the test coverage. In combination with detailed control flow and path analysis it might even be the case, that the resulting timing bound overestimates the real WCET. Consequently the acquired assurance is naturally higher than for end-to-end measurements. Nevertheless, the statements of [Butler and Finelli (1993)] also applies to hybrid approaches. Furthermore, the results cannot be proven to safely upper bound the WCET.

Static timing analysis is the only method that can claim to compute safe upper timing bounds, since a superset of all possible application states and input data is analysed. But it has to be ensured that the architecture model correctly abstracts the target hardware. For state of the art processors this naturally also requires testing on real hardware. With respect to abstraction of

hardware, differences between the timing formulae reported in the manual and those of the real hardware have been observed in [Atanassov et al. (2001)], illustrating the complexity and obstacles of designing an appropriate architecture model. Additionally, due to the reasons described in Section 2.4.2, static methods might suffer significant overestimation.

Probabilistic analysis methods, as an emerging alternative, perfectly fit the different failure rates of various DALs, since, according to the concept of probabilistic timing analysis, the WCET bound of an application can be reduced if lower failure probabilities are suitable. Unfortunately, since probabilistic analyses rely on randomized hardware, its applicability for COTS-focused systems is questionable.

Considering the individual drawbacks, especially the issue of sufficient test coverage, it is not possible to give a generally valid advice which timing analysis technique to use for which DAL. Rather, it depends on the overall system design, for example any implemented mitigation mechanisms such as redundancy concepts. Finally software is never certified in isolation, it is always considered as part of the whole system in combination with the target hardware. In the case of lower DAL it might also be sufficient to use for instance end-to-end measurements, as long as the achieved assurance is sufficient for the DAL.

2.5. Summary

In this chapter, topics that provide background information for this thesis, have been described. At first general terms have been defined. Afterwards the basic concepts of safety-critical systems and certification processes have been introduced. In this context, the partitioning concept also needed to be described since it clarifies the objectives of this thesis. It is important to note, that the definition of temporal partitioning is relaxed throughout this thesis, not referring to full isolation, rather than bounded interference. Finally, the state of the art in WCET analysis has been presented since the analysis approach of this thesis relies on the existing techniques.

3. Related Work

In the last years several approaches towards the integration of multi-core processors and real-time systems, especially addressing the topic of implicit resource sharing, have been proposed. Within this chapter they are roughly categorised as joint analysis, execution models, analysed/controlled resource sharing and dedicated hardware designs. In the following individual of these approaches are described in Section 3.1, 3.2, 3.3 and 3.4, respectively.

Since this thesis also addressed QoS aspects, examples of related approaches are described in Section 3.5.

3.1. Joint Analysis

To address the implicit resource sharing, joint analysis approaches analyse the mutual interactions between applications, that share the respective resource. Therefor, detailed knowledge on the state of each of the applications and the related processor core is required.

3.1.1. Shared Cache Analysis

The authors in [Yan and Zhang (2008)] apply a static WCET analysis to identify inter-thread conflicts on multi-core processors with shared level-2 (L2) instruction caches. Therefore, each level-1 (L1) cache miss is forwarded to the L2 cache analysis. If the L2 analysis indicates, that the access is a hit, its cache set number is determined and checked against the L2 cache usage of threads on other cores. If another core might use the same set, the access is classified as a miss. The classification of L2 accesses also accounts for instructions being executed in a loop or not. The approach has been evaluated against an all-miss strategy, revealing improved results.

In [Li et al. (2009)] and [Hardy et al. (2009)] the authors extend the approach of [Yan and Zhang (2008)]. In [Li et al. (2009)] message sequence charts and message sequence graphs are used to model the software system and identify threads with potentially overlapping life times. Hence, instead of accounting for the conflicts with all threads running on other cores, this allows to reduce the number of possible cache conflicts, since the number of concurrent threads is reduced. [Hardy et al. (2009)] use compiler techniques to reduce the number of potential conflicts. In particular, the authors control the cache content by only caching blocks, that are statically known to be reused.

3.1.2. Bus Analysis

As another integral part of multi-core processors shared interconnects, such as buses, need to be considered during analysis. Accordingly, the authors in [Chattopadhyay et al. (2010)] propose an integrated shared cache and bus analysis to account for the effect of the cache analysis on a shared bus. The shared cache analysis is based on inter-task conflicts and the individual task life times, as described in [Li et al. (2009)]. The bus analysis, also described in [Kelter et al. (2011)], assumes a TDMA-based arbitration to determine the worst-case access times and thereby the task

life time. Consequently the integration of shared cache and bus analysis is an iterative problem. In subsequent work [Chattopadhyay et al. (2012)], Chattopadhyay et al. extend their approach by additionally considering the effect of the shared cache and bus analysis on other parts of the processors pipeline, such as the branch prediction.

3.1.3. Statement

Overall, joint analysis approaches rely on in-detail, mutual analysis of in-parallel scheduled tasks. For that purpose flow information of all considered applications are required. This contradicts incremental certification and development, where modifications to one application shall not impact the development and analysis of other applications, since this requires significant analysis and certification overhead. Further, in huge projects, applications are often developed by different suppliers, hence implementation details are not freely available. Furthermore, the mutual analysis of applications entails increased complexity, since the detailed interactions, e.g. on shared caches, need to be analysed. Thus, the scalability of such approaches for an increasing number of applications and cores is questionable.

3.2. Execution Models

Approaches that propose an execution model deal with resource interferences by special restrictions for applications or parts of applications, which eases the analysis and allows the control of shared resource requests. Usually, this comprehends the division of applications into dedicated phases for computation and shared resource communication. In the following three examples are described.

3.2.1. Superblock Model

The authors in [Schranzhofer et al. (2009)] propose an execution model, that divides an application into a sequence of superblocks. Each superblock is subdivided into an acquisition, execution and replication phase. Each phase gets a maximum execution time and a maximum number of resource requests assigned. Based on this application model, the authors study the schedulability and the impact on the resulting execution time bound of different resource access models. They define the dedicated, the general and the hybrid access models. Thereby, the dedicated access model allows resource requests only during the acquisition and the replication phases. The general model does not rely on the defined superblock phases, instead it allows resource requests at any point in time. Finally, the hybrid model uses the division into different phases, but still allows resource requests in any of them. The shared resource arbiter is assumed to implement a Time Division Multiple Access (TDMA) access scheme, i.e. resource requests are separated by hardware.

Based on the evaluation results, the authors conclude that the dedicated access model should be applied to control the resource requests in multi-processor systems. According to this work, an application schedule, such as shown in Figure 3.1, would the consequence. As can be seen, due to the TDMA arbitration concurrent resource requests are avoided. Further, since the dedicated access model has been advised by the authors, the execution phase of each superblock depends on the pre-loaded data of the acquisition phase. Thus a core is idle until the data required for the execution phase is available in the local caches. In effect, a core is idle while it is not in the execution phase or other cores access the shared resource. Thereby, the amount of idle time increases with the number of cores.

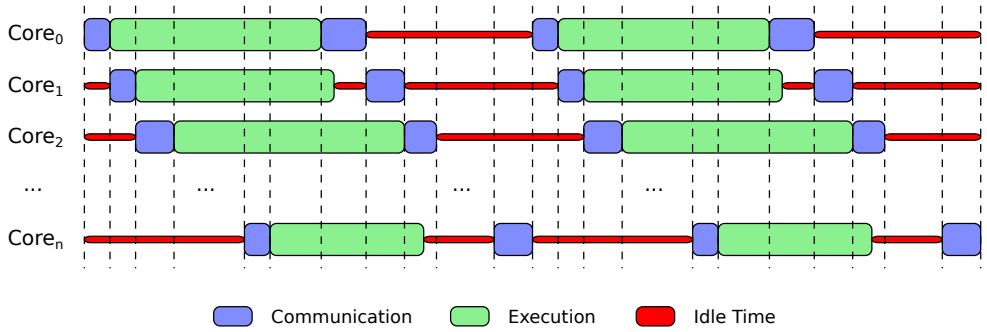


Figure 3.1.: *Resulting application schedule based on the conclusions of [Schranzhofer et al. (2009)].*

3.2.2. PREM Model

Another execution model, the PRedictable Execution Model (PREM), is presented in [Pellizzoni et al. (2011)]. It is targeted towards single-core processors and the isolation of concurrent request by the processing core and I/O devices. However, the authors believe that it is also applicable to multi-core processors.

The execution of an application is divided into a sequence of non-preemptible intervals, called predictable and compatible intervals. Predictable intervals are meant to form the largest portion of the application. They are subdivided into a memory and an execution phase. Comparable to the superblock model in [Schranzhofer et al. (2009)], the memory phase is used to pre-load required data and instructions into the local caches. Since no dedicated replication phase is defined, the memory phase is also used to update the main memory with the modified data from the previous execution phase. To prevent the access to applications-external code and data, the execution of operating system calls and interrupt handlers is prohibited. To still allow the execution of operating system code, the compatible intervals are defined. Beneath operating system calls and interrupt handlers, they are also used for the execution of code that is too complex to adhere to the restrictions of predictable intervals.

Based on this model memory requests of I/O devices are scheduled. To avoid any concurrency, I/O traffic is only allowed during the execution phase of a predicable interval. To guarantee that sufficient time for peripheral traffic is available, the length of each execution phase is enforced by execution time monitoring. Once an execution phase finishes before its defined bound, the core is kept idle. Hence, in essence the approach applies a serialisation of shared resource requests of the core and I/O devices.

The final adherence to the PREM model is achieved by compiler modifications. By the use of developer annotations and details on the cache hierarchy of the target platform, the code is restructured, such that the caches are pre-loaded and the monitoring of the execution phase length is added.

3.2.3. Sliced Execution

The model of sliced execution proposed in [Boniol et al. (2012)], splits the execution on a core in a sequence of alternating communication and execution slices. Similar to the model presented in [Pellizzoni et al. (2011)], execution slices perform only local computations without requests to shared resources. Accordingly, communication slices are used to pre-load caches and update main memory. The sequence of slices over all cores is defined by a static schedule. Unlike to the above described approaches, overlaps of communication slices between cores are not prohibited.

Instead it is assumed, that all required data are statically known. Based on that assumption, a network of timed automata is used to model the concurrent main memory requests. The modeling of those automata requires accurate information on the core-to-memory interface and the memory controller itself. If either of those information are not available or the required data are not statically known, the authors mention, that the maximum request latency needs to be applied.

3.2.4. Statement

In summary, execution models usually subdivide applications and assign restrictions with respect to shared resource requests to each fraction. Except [Boniol et al. (2012)], which relies on detailed architecture and application information, the main approach towards the sharing of resources is the complete avoidance of any concurrency either by assumed TDMA schemes or via higher-level software enforcement. Such resource privatisation is considered to inefficiently utilise the capabilities of modern multi-core processors, which often implement parallelism within resources, that cannot be leveraged if not more than a single request is issued at a given time. The drawbacks of such TDMA-like schemes have for instance been evaluated in [Kelter et al. (2013)]. As an effect, resource privatisation directly affects the processor core utilisation and the scalability of such approaches, considering the amount of idle times as illustrated in Figure 3.1. Further the assumption towards sufficient information on the architecture, as it is made in [Boniol et al. (2012)], might be suitable, but is at least questionable, considering the complexity of modern NoC-based multi-core platforms. Another general concern regards the relation of the length of communication and execution phases. That is, using core-local resources is somewhat intuitive, but if this would provide enough independence from external requests, the overall problem of shared resource interferences would not be as severe. Hence, it is assumed, that in contrast to the implicit assumptions of the presented approaches, execution phases are rather short compared to communication phases. Obviously, the actual relation depends on the individual application characteristics.

3.3. Analysed/Controlled Sharing

Joint analysis and the application of execution models are related to the work in this thesis insofar, that they address the same problem but with very different approaches. In contrast, the publications described in this section are closely related to this thesis, since they apply comparable solutions.

3.3.1. Occurrence-related Interference Analysis

In [Schliecker et al. (2010)] a method to compute the inter-task interference due to shared resources based a minimum distance between memory accesses is proposed. The authors focus on the computation of the maximum number of cache misses per core within a certain time interval, assuming preemptive scheduling. These bounds are further used to compute the inter-core interference and the response time of a task.

The work of [Schliecker et al. (2010)] is also considered by the authors of [Dasari et al. (2011), Dasari and Nelis (2012)]. They discuss a similar approach than Schliecker et al., but assume a non-preemptive task model. This allows some optimisations, which finally lead to tighter bounds on the number of cache misses per core. [Schliecker et al. (2010), Dasari et al. (2011), Dasari and Nelis (2012)] commonly consider online scheduling schemes, focusing on the computation of the

task response time. Further, they abstract the resource usage of applications depending on the temporal occurrence of requests.

Often development processes consist of design, implementation and integration phases. In the context of real-time systems, timing requirements are usually defined during the system design, while the adherence to those bounds is validated via timing analysis during the integration. It is considered very complex or even impossible to pre-define the resource usage of an application at design time, especially since the actual timing bound depends on the interactions with in-parallel scheduled applications. This inter-application dependence in the system development contradicts the requirements of incremental development and certification. Furthermore, since deviations between the defined and the actual bounds can only be detected during integration, it is also less efficient than approaches which allow the isolated analysis of applications, and thus an a priori compliance check during the development phase.

3.3.2. Resource Server

In [Yun et al. (2012)] a method to isolate the behaviour of multiple cores when accessing shared memory is discussed. They propose a server-based approach, that assigns a certain limit on the amount of cache misses. Thereby the authors focus on the isolation of one critical core from multiple non-critical ones by using one resource server per non-critical core. They parametrise the server limits based on bounds for the WCET and the resource usage of the critical core. For evaluation, the authors study and try to minimise the performance impact of the monitoring on the non-critical cores. Similar to [Schliecker et al. (2010), Dasari et al. (2011), Dasari and Nelis (2012)] they consider the specific occurrences of resource requests in time. However, they do not focus on how to determine the resource usage or the timing bounds.

A similar approach compared to [Yun et al. (2012)] is proposed in [Behnam et al. (2012)]. Instead of applying resource servers only to non-critical cores, Behnam et al. propose the usage of a hierarchy of servers for all cores.

3.3.3. Statement

The described approaches either apply a higher abstraction than joint analysis approaches to cope with the analysis complexity or try to control the inter-application interferences at runtime. Thereby they share the assumption of a uniform memory accesses, i.e. they rely on a single memory latency for their analysis. Therefor they either consider the maximum contention, which is very pessimistic or assume a linear latency increase, which contradicts measurements on real hardware as presented in Chapter 6. Further, some of the analyses require detailed information on in-parallel scheduled applications, which for one increases the complexity and also contradicts incremental development and certification. Finally, the mentioned resource server approaches do not account for runtime overheads of their server, i.e. additional execution time, resource requests and the impact on the application timing and resource boundaries.

3.4. Dedicated Hardware Designs

Another approach for solving concurrency issues in multi-core processors is a dedicated hardware design. This for instance includes modified resource arbitration policies and architectural solution, that provide isolation or increase predictability.

3.4.1. Interconnect Architecture

In [Rosen et al. (2007)] the authors discuss different bus access policies and related timing analysis. This includes the theoretically optimal schedule, a TDMA-based scheme and two improved TDMA-based solutions.

A further mean to cope with shared resource interference on the interconnection level is the design of the network. For instance, [Diemer and Ernst (2010)] proposes a NoC architecture that isolates traffic based on real-time requirements. In [Salloum et al. (2012)] a novel MPSoC architecture for safety-critical embedded systems is proposed, which also includes a deterministic interconnect.

3.4.2. Pipeline and Platform Analysis

The authors of [Paolieri et al. (2009)] propose a novel multi-core architecture to increase the analysability of real-time applications. They introduce a execution mode, called WCET computation mode, which artificially enforces maximum contention. This emulates the effect of the worst-case interference, as it would be imposed by concurrent tasks. As such the WCET computation mode can be used to analyse the WCET bound of applications without requiring information on in-parallel scheduled tasks. However, since this approach assumes maximum contention it is also very pessimistic. As another aspect of this work it is shown how the maximum contention, i.e. an upper bound on the interference, is determined. Therefor, the authors focus on shared caches and buses. The upper bound on the bus delay is discussed for different access policies, in particular priorities, round robin and first-in first-out. For the shared cache, interferences due to concurrent accesses as well as due to evictions of shared cache lines are considered. Concurrent requests are analysed similarly to the shared bus. For cache evictions different means for spatial partitioning are discussed.

3.4.3. Statement

Even though modifications to hardware provide interesting solutions and might be considered in the design of future platforms, they are infeasible for today's COTS systems. As explained in Section 1.1, this thesis is focused on COTS components, following the ongoing trends in industry. Usually, the designs of such processors cannot be modified and SoC designs additionally reduce this possibility due to the high integration level. For that purpose, dedicated designs are not further considered.

3.5. Quality of Service in Real-time Systems

While many related approaches solely focus on analysing the worst-case behaviour of applications, some also consider improved average-case performance.

3.5.1. Interconnect

For their NoC architecture, the authors in [Diemer and Ernst (2010)] present a QoS service extension. Unlike other NoC flow control schemes with real-time focus, the so-called back suction approach does not prioritise traffic with service guarantee requirements over best-effort traffic. Instead best-effort traffic is selectively preferred as long as the service guarantees are still met. Thereby, the best-effort latency could be improved significantly.

3.5.2. Scheduling

A commercially deployed approach to increase the average-case performance of hard real-time systems is constituted by slack scheduling, as it is used in the Digital Engine Operating System (DEOS) operating system, cf. [Binns (2001)]. It describes a preemptive fixed priority scheduling approach, which allows to accumulate unused processor time and assign it to enabled applications. In effect the overall system utilisation is increased.

3.5.3. Resource Server

In [Yun et al. (2013)] the authors present an extension for their previous work [Yun et al. (2012)]. The approach called MemGuard regulates the memory bandwidth towards the minimum service rate of the memory controller to increase the likelihood for immediate service. Since this is similar to an assumed full interference for all requests, the approach is very pessimistic. In addition to the isolation, the authors implement a prediction-based reclaiming mechanism, that allows for periodic adoptions of the allowed resource usage per core. However, since the mechanisms is prediction-based, false predictions can lead to situations where guaranteed bandwidth is not delivered. Hence, the proposed method is only applicable to soft real-time systems, which is a main differentiator to the work presented in this thesis. To address this limitation the authors indicate a selective disabling of the QoS extension without providing further details.

3.6. Summary

As shown, plenty of work on the implicit sharing of resources in multi-core processors has already been published. However, the identified drawbacks point out, that further research towards the practical deployment in mixed-criticality real-time systems is required. The main identified drawbacks are summarised as follows:

Incremental Approaches and Complexity Approaches that rely on detailed information on in-parallel scheduled applications contradict the goal of incremental certification and development. Further, the detailed information are often applied to mutual task analyses, which affects the complexity of the approach and hence renders the scalability for an increasing number of cores and applications questionable.

Mixed-criticality While most of the described approaches do not mention mixed-criticality requirements, others are not applicable due to the applied prioritisation between hard real-time or critical and soft real-time or uncritical applications.

Resource Privatisation Especially, execution models use resource privatisation to avoid resource sharing. However, as shown this greatly impacts system utilisation and should thus be avoided.

Uniform Access Latencies As described, the approaches that are more closely related to the work in this thesis commonly assume uniform memory latencies, either in the form the maximum latency or an assumed linear increase. While the former approach is very pessimist, the latter disregards arbitration latencies. As shown in Chapter 6, this assumption also contradicts measurements real hardware.

To address these drawbacks, the approaches presented in the following chapters apply a higher abstraction of applications, to no longer rely on detailed information of in-parallel scheduled applications. Beneath others, this shall help to maintain a complexity similar to the analysis of single-core processors, while still being able to increase the tightness of the multi-core timing bounds. Further, per se resource privatisation is avoided to leverage the parallel nature of multi-

core architectures with respect to analysis and runtime approaches. To address the requirements of mixed-criticality systems, the execution behaviour of all application shall be guaranteed. Finally, no assumptions on shared resource access latencies are made, except a fair arbitration policy to avoid starvation. Instead the actual platform parameters are determined and applied for timing analysis.

4. Temporal Partitioning and interference-sensitive WCET Analysis

Within this chapter the main concepts of the thesis are introduced. At first, the assumed system architecture is defined in Section 4.1. This comprises a generic multi-core model and the software architecture. Based on this architecture, the runtime resource usage enforcement and the isWCET analysis are introduced in Sections 4.2 and 4.3, focusing on the isolation of processor cores. In Section 4.4, the QoS extension is described. To cover all units of modern SoC architectures, Section 4.5 describes the integration of peripheral devices into the concepts. Finally, the integration into development processes is discussed in Section 4.6.

4.1. System Model

4.1.1. Abstract Multi-core Model

The term multi-core processor is commonly used for processor designs that integrate multiple processing cores, implementing the Multiple Instruction Multiple Data (MIMD) computing paradigm [Flynn (1972)]. It does however not specify any further details of the architecture, e.g. memory hierarchies, interconnection, integration of peripherals and design of processing cores. Instead, multi-core processors can be implemented with symmetric or asymmetric processor core designs, as host-like processors or SoCs, and using shared or distributed memory models.

Instead of covering all different implementation alternatives separately, an abstracted architecture model is used throughout the thesis. Since most of today's multi-core processors implement a shared-memory architecture, the model is based on such a design. Common elements of shared-memory multi-core architectures are Processing Elements (PEs) or cores, shared main memory, related platform caches and a NoC. In SoC-based platforms, also peripheral interfaces are integrated. Figure 4.1 summarizes the classification of different units within a SoC-based multi-core processor.

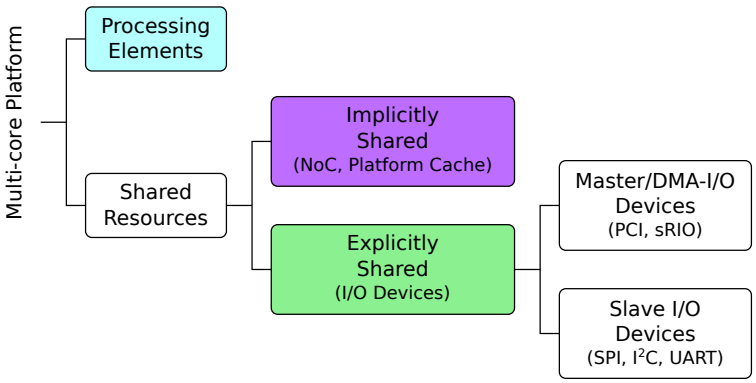


Figure 4.1.: *Classification of units within a SoC-based multi-core architecture.*

On the first level PEs and other resources are divided based on the fact, that only PEs execute user programmable software, while other resources are controlled by this software. The resources are further divided into implicitly and explicitly shared. Implicitly shared resources are not directly visible to the software layer, only their effect on execution timings can be observed. Hence, software does not directly communicate with such devices, rather than implicitly trigger accesses, for instance as the result of cache misses in the highest-level core-local cache. Prime examples for implicitly shared resources are the parts of the memory hierarchy in between PEs and main memory, i.e. the NoC and platform caches. The main memory itself is a special case, since naturally the SK configures the memory layout via the MMU, i.e. the sharing of the physical address spaces is under control of a software layer. However, the sharing is invisible to higher software layers, such as applications. Contrary to implicitly shared resources, accesses to explicitly shared resources are intentionally triggered by software. This usually applies to I/O devices. They are further grouped into master and slave devices. Master devices are capable of initiating accesses to memory via DMA, examples are network interfaces, Peripheral Component Interconnect Express (PCI/PCIe) and serial RapidIO (sRIO) devices. Hence, such DMA-capable Input/Output (DMA-I/O) devices can cause shared resource interferences to PEs. On the other hand, memory requests of slave I/O devices are solely triggered by software, this for instance applies to Universal Asynchronous Receiver/Transmitters (UARTs), Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) devices.

The described multi-core units are transitioned into the multi-core model ($S = (P, PE, R, IO)$) as shown in Figure 4.2 and described in the following.

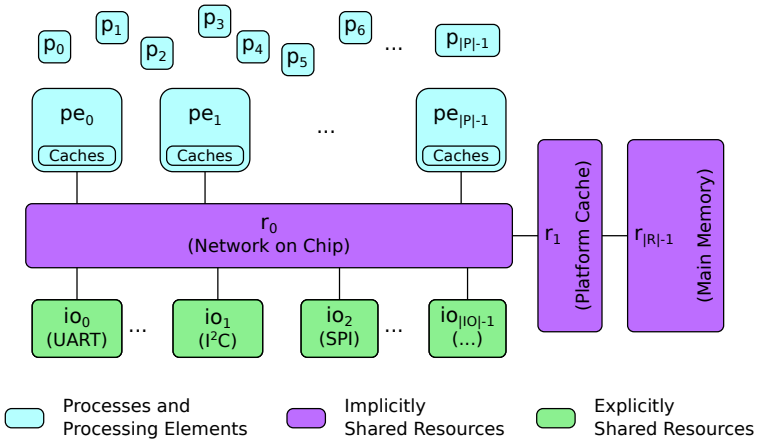


Figure 4.2.: *Abstracted shared-memory multi-core architecture model S.*

Definition 1 (Processing Elements (PEs) pe_j)**.** PEs abstract the concrete implementation of the target processor core. This includes all features that are known for single-core processors, e.g. the Instruction Set Architecture (ISA), pipeline implementation and local cache hierarchy. In order to cover symmetric as well as asymmetric designs the multi-core model allows different implementations for each of the PEs. The set of all PEs is referred to as $PE = \{pe_j\}, j \in \mathbb{N}_0, j = 0, .., |PE| - 1$.

For bounding the shared resource interference it is assumed, that a PE can only issue a fixed maximum number of requests per resource arbitration cycle. To ease the following explanations, only the case of one request per PE and arbitration cycle is described. However, in general, if PEs are able to issue more than one request at the same time, the additional requests can for instance be modeled as requests of artificial PEs. ■

Definition 2 (Implicitly Shared Resource r_k)**.** The implicitly shared resources of a system are represented as the set $R = \{r_k\}, k \in \mathbb{N}_0, k = 0, .., |R| - 1$. Each of these resources is considered to provide some kind of service with a given maximum *capacity*, such as bandwidth. Consequently, each resource is abstracted as its capacity c_{r_k} with $c_{r_k} \in \mathbb{N}_0$. If a single resource provides multiple services, that can be addressed independently, each of them is handled as a single shared resource. Furthermore, each resource can implement multiple channels. A channel is defined as part of a resource, that processes a single request. Hence, a resource with multiple channels can handle multiple requests in parallel. In contrast to services, channels are transparent for the system, i.e. they are not directly addressable and their existence can only be observed via resource access timings. To assign a request to a channel the resource has to implement an arbitration mechanism. It is defined, that channels are symmetric, i.e. it does not matter which channel serves which request. Hence, as soon as a channel becomes ready any outstanding request can be assigned to it. As an example, the main memory can consist of multiple separate memory controllers. Each of them is directly addressable, hence the memory resource provides multiple services, that can be handled as individual resources. On the other hand, each of the memory controllers might provide parallel interfaces to process several requests simultaneously. As long as these interfaces cannot be directly addressed, each of them is considered as a channel. For an analysis it is necessary to bound the access latency to shared resources, i.e. determine an upper bound. Therefore, starvation has to be prohibited, which requires a fair arbitration policy. ■

Definition 3 (Process p_i)**.** Naturally, PEs are under control of software scheduling, which assigns virtual PEs to physical PEs. Virtual PEs are further called *processes*. The set of all processes is defined as $P = \{p_i\}, i \in \mathbb{N}_0, i = 0, .., |P| - 1$. At a given time a single PE can at most execute one process. Accordingly, the set of in-parallel scheduled processes is defined as $P_{||} = \{p_i\}, i \in \mathbb{N}_0, i = 0, .., \left|P_{||}\right| - 1$ with $\left|P_{||}\right| \leq |PE|$, i.e. the number of in-parallel scheduled processes is limited by the number of available PEs.

Each process occupies a PE for a maximum time during which it can issue a maximum number of accesses to either core-local or shared resources. The maximum execution time is commonly referred to as WCET. Yet, the number of shared resource accesses has no defined terminology, hence the term *Worst-Case number of shared Resource Accesses (WCRA)* is defined. The WCRA can also be understood as a bound for the resource usage of the capacity of an explicitly shared resource. Respectively, a process is represented as a tuple $p_i = (t_{s,p_i}, \{c_{p_i,r_k}\})$, with $t_{s,p_i} \in \mathbb{N}$ being its core-local WCET bound and $c_{p_i,r_k} \in \mathbb{N}$ being its resource usage bound or capacity per utilised shared resource r_k. ■

Definition 4 (Explicitly Shared Resource io_l)**.** As shown in Figure 4.1, explicitly shared resources include master and slave I/O devices. Both of them are able to access implicitly shared resources, thus posing a potential threat for temporal partitioning. In contrast to PEs, explicitly shared resource implement a fixed set of functions without being subject to software level scheduling. Reprogrammable hardware, such as Field Programmable Gate Arrays (FPGAs), could be considered a special case, since their functionality can be changed at runtime. But still they are usually not covered by process scheduling. Unlike processes, explicitly shared resources are not covered by WCET analysis. Accordingly, they are abstracted as their WCRA bounds $c_{io_l,r_k} \in \mathbb{N}$ per utilised implicitly shared resource r_k. ■

4.1.2. Software Architecture

The software architecture is based on the partitioning concept shown in Figure 2.3. A SK operating system layer is in control of the underlying hardware, providing an interface to the software.

As described in Section 2.3 the SK provides a partitioned environment. This also includes partition/process scheduling in order to decide which process is assigned to which PE at a given time. The defined software architecture applies an offline scheduling scheme, i.e. the mapping between processes and PEs, and their activation times are known before execution. The SK implements an online dispatcher. The scheduling model is motivated by the time-triggered system defined by the ARINC 653 [ARINC (2003)] avionics standard. As described in Section 2.3.2, ARINC 653 defines a cyclic scheduling scheme, cf. Figure 2.5. Each process is represented by its period, duration and offset relative to the start of the scheduling cycle. The interval during which a process is active, is further called a *process frame*. The length of the scheduling cycle is called major time frame. During execution the dispatcher activates the processes as defined by the major time frame. A process switch can for instance be triggered by a timer. If a process has finished execution before its deadline, the processor is idle until the next process switch event occurs. Once the whole scheduling cycle has been executed the dispatcher starts over again with the first process.

The scheduling scheme used within this thesis sticks to the time-triggered model of ARINC 653 with its definition of process and major time frame. Likewise, the whole schedule is defined offline. Unfortunately, the current version of the standard does not cover multi-core scheduling. Hence it is defined, that the same major time frame is used over all cores. While the timing parameters, i.e. period, duration and starting times, are similar on all cores, the actually scheduled processes can be different. If the same process has multiple slots within the major time frame, each slot represents a new instance of that process. For that purpose, each instance starts from the process entry point, instead of continuing the execution of the last instance. Obviously, it is valid to assign a process to a process frame, whose duration is larger than the WCET bound of that process. Figure 4.3 illustrates the defined time-triggered multi-core scheduling.

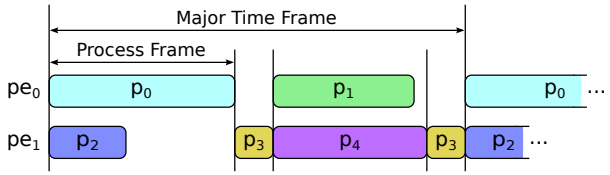


Figure 4.3.: *Applied multi-core scheduling scheme.*

4.2. Runtime Resource Usage Enforcement

The purpose of resource usage enforcement is to monitor the actual resource utilisation of the processes and enforce the adherence to their statically defined capacity. On system level this ensures temporal partitioning, since the maximum interference between concurrent requesters is bounded. As such, the monitoring is especially useful in mixed-criticality systems to provide an additional safety-net to protect higher critical applications against misbehaviour of lower critical applications. This is directly related to the different assumptions on the application behaviour, depending on its assurance level, cf. Section 2.2.

Within this chapter, mechanisms for temporal partitioning for PEs and implicitly shared resources are discussed. Explicitly shared resources are covered separately in Section 4.5, since they are naturally controlled via device drivers, which allows additional mechanisms that would exceed the focus of this section.

One should bear in mind, that the definition of temporal partitioning has been relaxed such, that it does not define full isolation of the temporal behaviour of applications, rather than a bounded temporal interference. As one aspect of temporal partitioning also fault containment is

guaranteed. In particular, in a non-partitioned environment the violation of the capacity could influence the resource access timings of other requesters, such that processes may suffer deadline miss and DMA-I/O devices cannot transfer their data in time. Contrarily, in partitioned system only the requester that causes the violation is affected, while all others are no further influenced than defined via the resource usage bounds.

Figure 4.4 illustrates the intended effect of resource usage enforcement, showing two cores, which are scheduled based on the defined multi-core scheduling scheme. The two processes p_0 and p_1 and their respective process frames within the major time frame are shown in details, while the remaining processes $p_2, .., p_{|P_{||}|-1}$ and their process frames are only indicated. The diagrams for p_0 and p_1 show the accumulated number of requests to an exemplary shared resource, considering different execution conditions. In particular, the continuous, green lines depict normal behaviour, while the dashed, red and dotted, blue lines illustrate abnormal and partitioned execution conditions, respectively. The resource usage bounds c_{p_i,r_k} of the two processes are marked with horizontal, dot-dashed, purple lines. The end of execution of either process is marked with an x, colored accordingly to the individual execution condition. Overall, only a single iteration of the major time frame is shown, while the same applies to all further iterations.

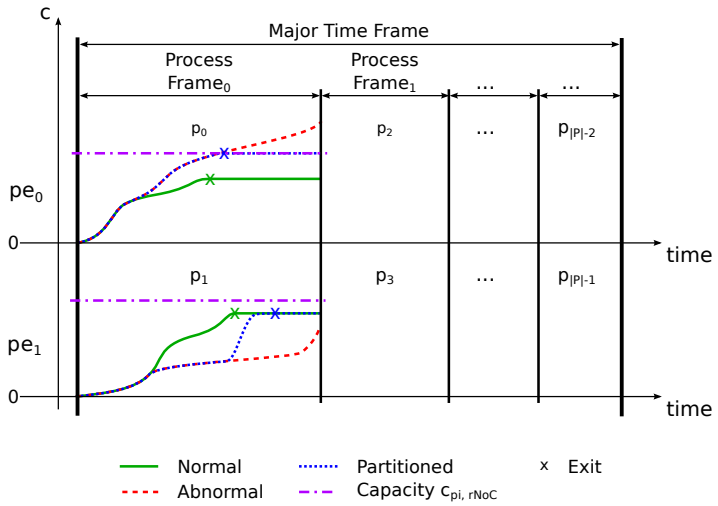


Figure 4.4.: *Resource usage enforcement example, showing the resource usage c of processes p_0 and p_1 over time. The different system conditions of normal, abnormal and partitioned behaviour are depicted via continuous, green; dashed, red and dotted, blue lines, respectively.*

As can be seen, under normal conditions both processes finish within their process frame. For abnormal conditions p_0 issues significantly more resource accesses than under normal conditions. This can for example occur due to a software error or an external effect, such as a Single Event Upset (SEU), hitting a register, which is used as a loop counter and thus increases the number of iterations and the likewise the number memory requests. In effect, p_0 executes until it is stopped at the end of the process frame. Its additional accesses interfere with those of p_1 in a way such, that p_1 suffers higher resource access delays while not being able to issue a sufficient number of requests. This finally causes a deadline violation. Contrarily, for a partitioned system, once p_0 exhausts its resource capacity, any further accesses to the shared resource are prohibited. In effect p_1 will still experience an increase in its execution time, but since a sufficient amount has been accounted offline, p_1 is able to finish within its deadline.

In order to provide a safe temporal partitioning mechanism the following properties have to be

fulfilled:

Temporal Transparency: The monitoring is an additional online mechanism, executed in parallel to the processes, hence it can potentially influence their temporal behaviour. The property of temporal transparency ensures, that a mechanism can be implemented such, that temporal influences on the process timing can either be completely avoided or bounded. In the latter case these bounds have to be incorporated into the process timing analysis.

Functional Transparency: Besides temporal behaviour, also the functional behaviour of the monitored processes shall not be affected. That is, the functional correctness of the process has to be retained. For example, the monitoring mechanism cannot utilise any monitoring facility, that is also used by any of the processes.

Additionally, also processes shall not be able to take control of, or affect the correct operation of the monitoring in any way. For instance, a process must not be able to disable the monitoring or alter the monitoring results, i.e. the observed resource usage.

The temporal partitioning mechanism can be divided into a monitoring and a suspension task. The monitoring is responsible to track the resource usage of all processes and continuously compare the actual usage with the respective usage bounds c_{p_i,r_k}. The exhaustion of c_{p_i,r_k} constitutes a capacity violation which has to be signaled to the suspension task. In effect, the suspension task has to immediately block a process from further accessing the affected resource to prohibit any additional interferences with in-parallel scheduled processes. In the next sections, implementation alternatives for both, monitoring and suspension are discussed, considering different state of the art processor features and software solutions.

4.2.1. Monitoring

Precise monitoring requires a monitoring facility that is able to identify which process utilises which resource. Therefor, the resource parameter(s) that are chosen to abstract the capacity, are required to adequately represent the resource. For instance, the bandwidth of a network. As already identified, the main sources of temporal interferences between processes in a multi-core processor are concurrent requests to implicitly shared resources on the path from a PE to the main memory. This comprises the NoC, shared caches, memory controller(s) and the main memory. Since accesses to individual of these resources cannot be distinguished by a process, they are further handled as one shared resource, referred to as *off-core memory*.

Definition 5 (Off-core Memory)**.** The off-core memory is defined as a shared resource, that abstracts all memory related parts between the PE and the main memory. Hence, the off-core memory covers the NoC, platform caches, the main memory controller(s) and the main memory. ■

From the perspective of a process, one of the most characteristical parameters to represent this off-core memory resource is its bandwidth. In general the bandwidth is the number of accesses within a certain time interval. Accordingly, the resource usage bound c_{p_i,r_k} represents the number of accesses of process p_i to the off-core memory. Thus, the monitoring is required to determine the number of memory accesses over a certain time. In the following a fully software-based and a hardware-assisted approach are described.

Software-Based Monitoring

A fully software-based approach requires modifications to the process binary, which track the resource requests and compare the total amount of accesses against the defined capacity of the

process. Typical alternatives are automated instrumentation by the compiler and manual user annotations within the source code. The number of off-core memory requests, beneath others, depends on load/store instructions as well as the addresses of instruction and the overall cache behaviour of the code. For instance compiler optimisations can significantly reorder instructions and the cache behaviour can only very hardly be predicted during software development. Accordingly, it is very hard or even impossible for a programmer to predict the number of memory requests. Thus, in the following only automated instrumentations are considered applicable.

To determine the number of memory accesses for a given program, each load/store type instruction has to be considered. This includes data load/store instructions, instructions fetches, cache operations like flushes, and any other instructions which, for the given architecture, are defined to trigger off-core memory requests. The instrumentation code has to increment a counter and compare its value against the resource usage boundary c_{p_i,r_k}. Should the counter exceed c_{p_i,r_k}, the suspension routine needs to be called. To reduce the overhead of additional memory accesses by the instrumentation code, the request counter, c_{p_i,r_k}, as well as the instrumentation code should be held in core local resources, e.g. the caches or registers. Furthermore, changes to the processor state, e.g. changing the condition register by executing a compare instruction, have to be handled.

An exemplary code snipped for the PowerPC architecture is shown in Listing 4.1. In lines 2 to 6, the registers r13, r14 and the condition register are saved to the stack. Afterwards the resource counter as well as the resource capacity are loaded from memory, the counter is incremented and compared against the capacity, while the suspension routine is called if the counter value is larger than the capacity (lines 9 to 13). Finally, the stack content is restored to the respective registers in lines 16 to 20.

```
1  /* create stack frame based on stack pointer r1 */
2  stwu    r1, -0x10(r1)          // increase stack
3  stw     r13, 0x4(r1)           // save r13
4  stw     r14, 0x8(r1)           // save r14
5  mfcr    r13                    // copy condition register
6  stw     r13, 0xc(r1)           // save condition register
7
8  /* actual monitoring */
9  lwz     r13, 0x0(<counter>)    // load resource counter
10 lwz     r14, 0x0(<capacity>)   // load resource capacity
11 addi    r13, r13, 0x1          // increment load/store counter by 0x1
12 cmpw    r13, r14               // compare counter against boundary
13 bgt     <suspension>           // call suspension routine if r13 > r14
14
15 /* restore registers from stack */
16 lwz     r13, 0xc(r1)           // restore condition register
17 mtcr    r13
18 lwz     r14, 0x8(r1)           // restore r14
19 lwz     r13, 0x4(r1)           // restore r13
20 addi    r1, r1, 0x10           // decrease stack
```

Listing 4.1: *Instrumentation code to increment memory accesses counter (register r13) and branch in case it exceeds the resource usage bound c_{p_i,r_k} (register 14).*

Facing the overhead of the instrumentation, a more coarse grained monitoring level than individual instructions, should be chosen. Based on state of the art code analysis approaches the basic block level, cf. Section 2.4, is considered appropriate. Each basic block is instrumented with the code of Listing 4.1, but instead of incrementing the counter by 0x1, it is incremented with a variable value, depending on the individual basic block. To determine the exact increment

per basic block different methods can be applied. In the simplest case the compiler counts the number of load/store type instructions, including instruction fetches. Since this approach does not account for the cache behaviour of the basic block, it would greatly overestimate the number of off-core accesses. As an alternative, to increase the tightness of the increment value, timing analysis techniques as described in Section 2.4 can be applied. To reduce the impact on the code size, the instrumentation can also be implemented as a function that is called at the beginning of each basic block. The function uses a lookup table, that maps return addresses to the number of load/store type instructions. In effect, the spatial overhead of the instrumentation would be reduced to a single branch per basic block. However, the temporal overhead for executing the instrumentation code still remains.

As defined above, a safe monitoring has to fulfil functional and temporal transparency. Functional transparency is not violated, although the initial code is modified, its semantics is not affected. Hence the results are still correct. Likewise, temporal transparency is fulfilled. Even though the temporal behaviour of the process is changed by executing additional instructions, its effect can be bounded since the instrumentation is part of the resulting binary.

Hardware-assisted Monitoring

An alternative to software instrumentation are hardware-assisted approaches. Typically, processors implement so-called performance monitoring facilities, e.g. in the form of processor core performance counters and platform wide debug and performance monitoring units.

Core-local performance counters provide the ability to assess the behaviour of different core functional units like caches, MMU and other pipeline stages by counting related events. For instance, the caching behaviour of a process can be characterised via the total number of cache accesses versus the number of suffered cache misses. Typical performance measures for pipeline stages are the number of completed instructions and stall cycles over a certain time. Such performance counter facilities can for example be found in the PowerPC [IBM (2010)], ARM [ARM (2010)] and x86 [Intel (2013)] architectures. Usually, such counters increment each time the configured event occurs. Optionally, they also issue an exception once a specified value has been reached or the counter detects an overflow. Hence, an appropriate exception can be used to signal the exhaustion of the resource capacity and trigger the suspension routine. If the counters do not provide an exception mechanism, polling implementations need to be applied. In practice polling implementations should be avoided, since they require periodic interruption of the monitored processes, which complicates timing analysis. Furthermore, polling requires a certain interval, i.e. capacity violations can only be detected at interval boundaries. This reduces the precision compared to an exception-based method, where each individual access is checked for a violation. To specifically monitor the usage of the off-core memory, all events that are related to the memory hierarchy of the core can be of use, e.g.:

- the total number of issued and completed load/store instructions,
- the number of cache accesses versus the number of hits and misses,
- the number of accesses to the interface between PE and NoC

In [Bellosa (1997a), Bellosa (1997b)], the author discusses the usage of cache-related information to deduce external memory accesses. In general this is a valid approach, only accesses that bypass the cache, such as non-cacheable accesses, have to be handled differently.

Aside from counters within the PE, modern SoC-based designs provide further monitoring facilities. Due to the increased processor integration, events such as bus traffic, that could be observed outside of the physical chip in former architectures, are now fully contained within the

silicon and thus invisible at first glance. In order to still provide sufficient visibility to developers, the manufacturers had to integrate advanced monitoring and debug features. Examples for such implementations are the ARM CoreSight architecture [ARM (2009)] and the Nexus [IEEE-ISTO (1999)] implementation within the Freescale P4080 [FSL (2012)]. Those debug facilities also include features, that allow the tracing of NoC and memory accesses, and the like. For more information, [Hopkins and McDonald-Maier (2006)] provides a review on features of today's debug architectures.

Considering SoC-level monitoring facilities, potentially each individual unit of the off-core memory resource can be used to acquire the process capacity usage, but might not be equally suitable. For example, the main memory controller will never recognise accesses, that are already processed in the platform cache. While such accesses still can cause interference on the NoC and cache they would not be considered if only main memory controller events are used. Likewise, not every NoC access is necessarily visible to the platform cache, e.g. NoC accesses that require a retry due to a detected collision would never reach the cache controller. The same applies for communications to I/O devices. For that purpose, the NoC is the most reasonable place of SoC-wide units to acquired off-core memory access information. As described above, since the SoC-wide units monitor all PEs at once, they are required to precisely distinguish them. This particularly requires sufficient counting facilities, which in fact is very architecture and implementation dependent.

To signal a capacity violation, a similar exception-based or polling mechanism, as for core-local performance counters, can be applied.

The transparency properties of hardware-assisted monitoring depends on the actual implementation in hardware. Usually, performance counters on core and SoC level operate independent from the processor pipeline and connected devices in the SoC. Once this can be proven for the target platform, temporal transparency is fulfilled. In the case of polling-based implementations, the periodic preemption of the program flow, as well as the additional execution time influence the timing of a process. To still ensure temporal transparency, both effects need to be covered during the timing analysis of processes. In order to guarantee functional transparency, it has to be ensured, that processes cannot manipulate the performance counters, e.g. by resetting the respective register. In general, the processes are not allowed to anyhow use the related performance counters during normal execution, for instance to acquire information on their performance. Furthermore, the correct handling of capacity violations has to be ensured. That is, for exception-based mechanisms, a process must not suppress the related exceptions. Similarly, for polling-based mechanisms, the periodic event that triggers the polling, must not be suppressed.

4.2.2. Suspension

The suspension is invoked once a capacity violation has been detected. As such, it is responsible to prohibit any further usage of the affected resource by the respective process for the remaining time of the current process frame.

Theoretically, it is sufficient to prohibit the usage of the affected resource by a process, e.g. through avoiding NoC accesses, while still allowing computations on local resources. Whether such fine grained access control is viable in practice depends on the architecture and resource implementation, for instance if it is possible to disconnect PE and NoC. Further, it depends on the functional necessity of the affected resource, i.e. whether or not the resource is essentially required for the process to fulfil its functionality. For example, the NoC is an essential resource, since at some point in time each process has to communicate its results somewhere outside of the PE, might it be to memory or a controlled I/O device. Additionally, since capacity violations indicate abnormal behaviour, the likelihood for the process to produce correct results is questionable.

Depending on the conditions in the target system the suspension can be implemented differently. In general, it is possible to suspend a process entirely and prohibit any further interference. If software instrumentation is applied for the monitoring task, the process can be suspended via a defined function or system call. For hardware-assisted monitoring, either the polling mechanism or the exception handler has to invoke a similar function or system call. Accounting for the synchronisation on process frame level, a process has to be suspended until the end of the current process frame to avoid further interference with the remaining processes. Once all processes of the current frame have finished execution and it is still time left till the end of the process frame, all suspended processes can be re-activated, in order to allow them to progress and even finish their task.

Since the suspension task is intended to influence the process behaviour, the properties of temporal and functional transparency do not apply.

In addition to the implementation of the suspension, also the implications for the system need to be considered. As described, a process is suspended once it exhausts its resource capacity. From system perspective, a suspension equals a deadline miss, in which case a process is not able to deliver its result within the given timing and resource constraints. Therefore, the suspension action always has to be handled on a higher system level to account for the exceptional situation and the impact on the overall system functionality. Such mitigation mechanisms are usually referred to as error handling strategies and include mechanisms like redundancy and check-pointing concepts, system reset or logging of the occurred event for maintenance purposes. However, the particular technique depends on the actual system and the criticality of the affected processes.

4.3. Interference-sensitive WCET Analysis

As described in Section 2.4, WCET analysis is a method to determine the execution time demand of a process. The analysis of single-core processors is a well understood problem and even architectures with complex pipelines and caches can be analysed. The main issue for the timing analysis of multi-core processors is shared resource interference, causing unpredictable behaviour for individual accesses, e.g. in terms of latency variations. In particular, the latencies of individual instructions no longer solely depend on the respective application, but rather are influenced by code that is executed on other PEs. In this section, the concept of the isWCET analysis is introduced to address the shared resource interference, while considering the drawbacks of related approaches as discussed in Chapter 3. In particular, the isWCET analysis is design to avoid full resource privatisation, while providing a scalable approach. The analysis relies on the abstract multi-core model, defined in Section 4.1.1, and the enforced resource capacities, as introduced with the monitoring concept in Section 4.2. The intuitive idea is to split timing analysis into core-local time and resource analyses and complement them with the computation of the shared-resource interference-delay to compute the overall isWCET bound. This poses two new challenges to timing analysis. Firstly, the resource usage bounds c_{p_i,r_k} need to be determined. Secondly, the maximum inter-process interference needs to be computed and incorporated with the core-local timing. For that purpose the following explanations are divided into the description of the core-local analysis phase and the two novel phases of interference-delay analysis and isWCET bound computation.

4.3.1. Core-local Analysis

The core-local analysis phase is used to determine the core-local WCET bound t_{s,p_i} and the WCRA bounds c_{p_i,r_k} for every process p_i and implicitly shared resources r_k. Due to their close relation it

makes sense to use similar techniques to determine the bounds for WCET and WCRAs. The state of the art for timing analysis has already been described in Section 2.4. In the following the three main approaches of end-to-end measurements, static analysis, and hybrid measurement-based analysis are considered. For static analysis and hybrid techniques, the described architecture for timing analysis, cf. Figure 2.7, comprising control flow construction and analysis, micro-architecture analysis, and path analysis, is assumed. In the following, modifications and extensions, required for shared resource analysis are described.

End-to-End Measurements

To acquire a timing bound with end-to-end measurements, the execution time over the whole process execution is measured. Similarly, the resource usage has to be recorded. Therefor, measurements as described for hardware-assisted runtime resource monitoring can for instance be applied. The monitoring facility is initialised at the start of each measurement and its status is read once the measurement has finished. Since the worst-case resource usage might be triggered with different input data than the worst-case timing behaviour, the set of input data vectors used for timing measurements needs to be adapted.

Static and Hybrid Measurement-Based Analyses

The control flow construction and control flow analysis phases of static and hybrid analyses solely rely on the binary and the set of input data, without considering timing parameters. On the other hand the micro-architecture analysis as well as the final path analysis assign timing information to each basic block and compute the WCET bound. For that purpose, control flow construction and analysis remain the same for timing and resource analysis, while the micro-architecture analysis and path analysis need to be extended.

During WCET analysis, the micro-architecture analysis determines bounds for the execution time of each basic block, either based on static analysis or measurements. In a similar way, resource usage information have to be annotated. The static timing analysis considers the execution time of instructions, their mutual interactions within the pipeline and the cache state. For resource analysis, the number of resource accesses is determined based on the number load/store type instructions as well as the cache state. Therefor, already available data, such as the cache hit/miss classification of memory accesses can be used to distinguish core-local and off-core memory accesses. To safely upper-bound the analysis, accesses that are classified as unknown have to be handled as cache misses, i.e. off-core accesses. In contrast to static analysis, hybrid approaches rely on measurements to annotate the timing information. To additionally determine the resource usage, either the software or hardware-assisted monitoring solution, cf. Section 4.2, can be applied.

The purpose of the path analysis is to determine the longest execution path through the control flow representation, based on the flow facts and the basic block timings. An extension towards the computation of the overall WCRA bounds requires to solve a similar optimisation problem. But instead of optimising for the longest execution time, the optimisation criteria is the highest amount of shared resource accesses.

As there may be more than a single shared resource, the core-local resource analysis has to be preformed for each resource. As described, the computation of the timing bound does not need to be modified. Consequently, the resulting timing bounds already account for resource access latencies, since they are either measured or determined as part of the architecture analysis. Given that the isWCET analysis is based on separation of core-local and interference-delay analysis, the

already included resource latencies need to stripped from the core-local timing bound to avoid a double consideration and related overestimations. For end-to-end measurements, the resource usage is recorded for every measurement, irrespectively if the input data is targeted towards worst-case timing or resource behaviour. Likewise, for static and hybrid analyses, the path analysis has to compute the number of resource requests on the identified worst-case timing path(s) additionally. Applying this approach to timing analysis, additionally yields a resource usage bound for the computed worst-case timing path. If multiple paths lead to the same worst-case timing boundary, the minimal resource bound over all of those paths has to be considered to finally obtain a safe bound. To remove the resource access latencies from the core-local timing bound, the number of requests on the respective path is multiplied with the minimal access latency and subtracted from the timing bound. It is essential to use minimal latencies to not violate the correctness of the approach. Since the resource bound computed during resource analysis will, by definition be greater or equal than the resource bound on the worst-case timing path and further always accounted with at least minimal access latency, it is safe to subtract the requests as described.

4.3.2. Interference-delay Analysis

The interference-delay analysis is the second phase of the isWCET analysis. It computes the inter-process interference based on the WCRA bounds of in-parallel scheduled processes. In the following, the analyses of single-channel and multi-channel resources are discussed separately.

Single-Channel Resources

As described above, inter-process interference stems from the concurrent usage of shared resources, i.e. the combination of sequentially and in-parallel issued requests to resources, which leads to unpredictable latencies for individual accesses. The reason for latency dependencies between processes are limited bandwidth of the resource and additional arbitration delays, caused by the necessity to resolve concurrent accesses. For example, a shared resource with a limited number of read and write channels has to arbitrate between requesters once the number of requests is higher than the number of available channels. In the case of two requests and a single channel, the arbiter has to assign the channel to one of the requesters. Therefore, the not-arbitrated requester is delayed until the selected one has been finished. Based on the assumed fair arbitration, cf. Section 4.1.1, these interferences are modeled as follows. Considering a sequence of resource accesses, its worst-case execution time will constantly increase, the more additional requests are issued in parallel, i.e. the relative access latency increases with the number of requesters.

Definition 6 (Access Lantecy)**.** Access latencies are defined as $d_i \in \mathbb{N}$, whereat i indicates the number of parallel requests. ■

According to Definition 6, the worst-case latency for a single request is defined as d_1, for two parallel requests d_2 and so forth. Since the number of requests is bounded by the number of in-parallel scheduled processes $P_{||}$, $d_{\left|P_{||}\right|}$ constitutes the maximum latency.

Definition 7 (Single-channel Access Latency)**.** The relation between access latencies d_i and d_{i+1} for single-channel resources is defined as:

$$d_{i+1} = \underbrace{\frac{i+1}{i}}_{\text{contention}} \cdot d_i + \underbrace{a_{i+1},}_{\text{arbitration delay}} \quad \begin{array}{l} i \in \mathbb{N}, i = 1, .., \left|P_{||}\right|, \\ a_{i+1} \in \mathbb{N}_0 \end{array} \tag{4.1}$$

Thereby, the factor to d_i indicates the contention caused by limited bandwidth, while a_{i+1} represents the respective arbitration delay for $i+1$ requests. Arbitration delays depend on the actual resource implementation. Thus, a_i is considered, either as constant factor with $a_i = a_{i+1}\ \forall i$, or dependent on the number of concurrent requests i. ■

Since the arbitration delay a_i cannot be negative, Equation 4.2 describes the relation between relative access latencies $\frac{d_i}{i}$ for i and $i+1$. In consensus with the above argumentation, the relative delays increase with the number of requests. Consequently, the request latency will never decrease, the more requests are issued in parallel, since no two i and $i+1$ exist where $\frac{d_i}{i} > \frac{d_{i+1}}{i+1}$ applies.

$$\frac{d_i}{i} \leq \frac{d_{i+1}}{i+1}, \quad i \in \mathbb{N}, i = 1, .., \left|P_{||}\right| \tag{4.2}$$

Since interference stems from concurrent requests to a shared resource, the inter-process interference problem can also be formulated as the worst-case overlap of resource requests. In typical programs, off-core memory accesses are interleaved with computational instructions. Depending on the resulting interleaving of off-core memory requests between in-parallel scheduled processes, the probability for collisions on the off-core resource is affected. In particular, compared to a hypothetical process that solely issues off-core memory requests, interleaved computations reduce the probability for off-core memory collisions with other processes. Since for the WCET analysis only the worst-case is deciding, computational instructions are disregarded during the interference-delay analysis. This means, all c_{p_i,r_k} off-core requests of process p_i are handled as a single block of consecutively issued requests.

Definition 8 (Request Block)**.** A request block for a process p_i is defined as the sequence of its c_{p_i,r_k} resource requests, that are issued to a resource r_k. ■

Condition 1. The relation between the number of requests for the requests blocks of processes p_m and p_n is described as: $c_{p_n,r_k} \leq c_{p_m,r_k}$. ♦

Definition 9 (Overlap Scenarios)**.** The following three overlap scenarios for the request blocks of p_m and p_n are defined:

Sequential Overlap: For sequential overlap the request blocks are scheduled back-to-back, cf. Figure 4.5. The corresponding interference-delay t_{seq} is computed according to Equation 4.3.

Partial Overlap: Partial overlapping request blocks interfere by c_{ov} requests, cf. Figure 4.6. Accordingly, the interference-delay t_{prt} is computed as shown in Equation 4.4. The amount of overlapping requests is described by c_{ov}, with $0 \leq c_{ov} \leq c_{p_n,r_k}$.

Parallel Overlap: For parallel overlap, as many requests of both blocks as possible overlap, cf. Figure 4.7. According to Condition 1, the request blocks for p_m and p_n cannot overlap by more than c_{p_n,r_k} requests. Thus Equation 4.5 describes the respective interference-delay t_{pll}.

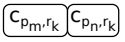


Figure 4.5.: *Sequential overlap.*

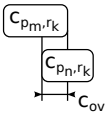


Figure 4.6.: *Partial overlap.*

Figure 4.7.: *Parallel overlap.*

$$t_{seq} = d_1 \cdot (c_{p_m,r_k} + c_{p_n,r_k}) \tag{4.3}$$

$$t_{prt} = d_1 \cdot (c_{p_m,r_k} + c_{p_n,r_k} - 2 \cdot c_{ov}) + d_2 \cdot c_{ov} \tag{4.4}$$

$$t_{pll} = d_1 \cdot (c_{p_m,r_k} - c_{p_n,r_k}) + d_2 \cdot c_{p_n,r_k} \tag{4.5}$$

■

Theorem 1. *For a pair of requests blocks, complying to Condition 1, the interference-delay for partial overlap is always bounded by the interference-delays for sequential and parallel overlap.*

Proof. The extreme values for the interference-delay of partial overlap t_{prt} are determined by $c_{ov} = c_{p_n,r_k}$ and $c_{ov} = 0$. Applying the extreme value for $c_{ov} = c_{p_n,r_k}$ to Equation 4.4 yields:

$$\begin{aligned} t_{prt_{c_{ov}=c_{p_n,r_k}}} &= d_1 \cdot (c_{p_m,r_k} + c_{p_n,r_k} - 2 \cdot c_{p_n,r_k}) + d_2 \cdot c_{p_n,r_k} \\ &= d_1 \cdot (c_{p_m,r_k} - c_{p_n,r_k}) + d_2 \cdot c_{p_n,r_k} \\ &= t_{pll} \end{aligned} \tag{4.6}$$

Likewise, Equation 4.4 can be simplified for $c_{ov} = 0$:

$$\begin{aligned} t_{prt_{c_{ov}=0}} &= d_1 \cdot (c_{p_m,r_k} + c_{p_n,r_k}) \\ &= t_{seq} \end{aligned} \tag{4.7}$$

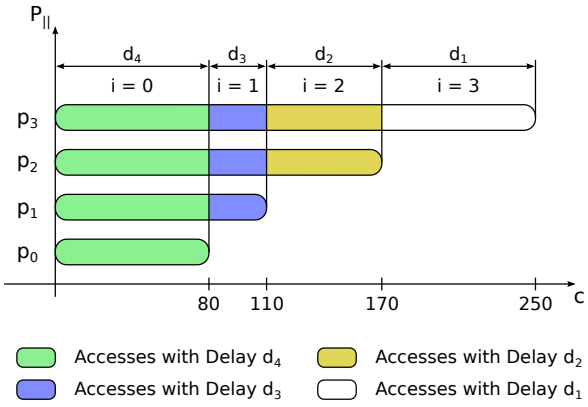
Accordingly, t_{prt} is bounded by t_{seq} and t_{pll}. □

Conclusion 1. *Since Theorem 1 is true for every pair of processes p_m, p_n and their respective resource capacities c_{p_m,r_k}, c_{p_n,r_k} for a resource r_k, it is safe to disregard partial overlaps during the analysis of the worst-case overlap. Instead, it is sufficient to rely on sequential and parallel overlaps. As a consequence the number of possible combinations of overlaps for multiple processes is drastically reduced, since the different cases, which arise due to the combination of c_{ov} potentially overlapping request, can be disregarded.*

Since Theorem 1 has been proven for an arbitrary pair of processes, the proof iteratively applies to all pairs of c_{p_m,r_k}, c_{p_n,r_k}. However, in order to compute the worst-case overlap for an arbitrary number of processes a generic formulation for t_{seq} and t_{pll} is required.

Definition 10 (Request Ordering)**.** For an arbitrary number of processes p_i, their resource capacities c_{p_i,r_k} are sorted in ascending order, described as: $c_{p_i,r_k} \leq c_{p_{i+1},r_k}\ \forall p_i, p_{i+1} \in P_{||}$. ■

For the parallel overlap scenario, Figure 4.8 depicts an example for $\left|P_{||}\right| = 4$ processes. The x-axis shows the number of request c for the request blocks of the processes p_0 to p_3. The respective processes are shown on the y-axis. The different colors indicate which delay d_i is applied for which amount of requests.



$$\begin{aligned} t_{pll}(p_3, r_k) = \quad & d_4 \cdot 80 \\ & + d_3 \cdot 30 \\ & + d_2 \cdot 60 \\ & + d_1 \cdot 80 \end{aligned}$$

Figure 4.8.: *Parallel overlap and interference-delay $t_{pll}(p_3, r_k)$ computation example for $\left|P_{||}\right| = 4$ processes and respective capacities c, illustrating the applied latencies d_i.*

Based on the runtime monitoring described in Section 4.2, the worst-case resource accesses for all processes are bounded by their capacities c_{p_i,r_k}. For that reason, two processes p_m, p_n cannot interfere by more than $Min\left(c_{p_m,r_k}, c_{p_n,r_k}\right)$ accesses. This complies with Equation 4.5, where c_{p_n,r_k} constitutes the minimum over c_{p_m,r_k} and c_{p_n,r_k}. Considering the example in Figure 4.8, all accesses of p_0 potentially interfere with the accesses of all other processes. Hence, the respective access delay is the maximum delay d_4. If p_0 exhausts $c_{p_0,r_k} = 80$ accesses, the remaining accesses of the other processes will at most experience a delay of d_3. The number of accesses of p_1, that can be accounted with reduced delay can be expressed as the difference of $c_{p_1,r_k} - c_{p_0,r_k} = 30$. Similarly, the accesses with reduced delays can be computed for the remaining processes.

To compute the interference-delay $t_{pll}\left(p_h, r_k\right)$ for process p_h and resource r_k, Equation 4.5 is extended to Equation 4.8. Definition 10 is essentially required.

$$t_{pll}\left(p_h, r_k\right) = d_{\left|P_{||}\right|} \cdot c_{p_0,r_k} + \sum_{i=1}^{h} \left[d_{\left|P_{||}\right|-i} \cdot \left(c_{p_i,r_k} - c_{p_{i-1},r_k}\right) \right] \tag{4.8}$$

According to Definition 9, sequentially issued requests are assumed to be processed without interference by other requests, i.e. the applied request latency is equal to d_1. Therefore, the overall interference-delay $t_{seq}\left(p_h, r_k\right)$ for process p_h and resource r_k can be computed according to Equation 4.9.

Condition 2. For sequential overlap, requests of processes with lower indices are issued before the requests of processes with higher indices. ♦

Condition 2 is essential to have a common basis for the comparison of t_{seq} and t_{pll}, hence it is assumed for the computation of $t_{seq}\left(p_h, r_k\right)$.

$$t_{seq}\left(p_h, r_k\right) = d_1 \cdot \sum_{i=0}^{h} c_{p_i,r_k} \tag{4.9}$$

Theorem 2. *The maximum interference-delay for single-channel resources is upper-bounded by* $t_{pll}\left(p_h, r_k\right)$*, i.e.* $t_{seq}\left(p_h, r_k\right) \leq t_{pll}\left(p_h, r_k\right)$.

Proof. To prove Theorem 2, it is shown that it still holds if Equation 4.8 is lower-bounded. Therefor, Equation 4.1 is first transformed to express d_i in dependence of d_1. Applying the series expansion to Equation 4.1 yields Equation 4.10.

$$d_i = i \cdot d_1 + \sum_{j=2}^{i} \left(\frac{i}{j} \cdot a_j \right) \tag{4.10}$$

Equation 4.8 can be lower-bounded by removing the second part of the equation. This is the case if all processes have the same resource capacity, i.e. Equation 4.11 is true.

$$c_{p_i,r_k} = c_{p_{i+1},r_k} \quad \forall p_i, p_{i+1} \in P_{||} \tag{4.11}$$

Applying Equation 4.11 to t_{pll} (Equation 4.8) yields Equation 4.12

$$t_{pll}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} = d_{\left|P_{||}\right|} \cdot c_{p_0,r_k} \tag{4.12}$$

Similarly, Equation 4.11 has to be applied to t_{seq} (Equation 4.9):

$$t_{seq}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} = d_1 \cdot (h+1) \cdot c_{p_0,r_k} \tag{4.13}$$

Further, $d_{\left|P_{||}\right|}$ can be substituted in Equation 4.12 according to Equation 4.10:

$$\begin{aligned} t_{pll}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} &= \left[\left|P_{||}\right| \cdot d_1 + \sum_{j=2}^{\left|P_{||}\right|}\left(\frac{\left|P_{||}\right|}{j} \cdot a_j\right)\right] \cdot c_{p_0,r_k} \\ &= \left|P_{||}\right| \cdot c_{p_0,r_k} \cdot d_1 + c_{p_0,r_k} \cdot \sum_{j=2}^{\left|P_{||}\right|}\left(\frac{\left|P_{||}\right|}{j} \cdot a_j\right) \end{aligned} \tag{4.14}$$

The relation between t_{seq} and t_{pll} under the condition of Equation 4.11 finally yields:

$$\begin{aligned} t_{seq}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} &\stackrel{!}{=} t_{pll}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} \\ d_1 \cdot (h+1) \cdot c_{p_0,r_k} &\stackrel{!}{=} \left|P_{||}\right| \cdot c_{p_0,r_k} \cdot d_1 + c_{p_0,r_k} \cdot \sum_{j=2}^{\left|P_{||}\right|}\left(\frac{\left|P_{||}\right|}{j} \cdot a_j\right) \end{aligned} \tag{4.15}$$

Since $h = 0, .., \left|P_{||}\right| - 1, c_{p_i,r_k} \geq 0$ and $a_i \geq 0$:

$$\begin{aligned} 0 &\leq \left(\left|P_{||}\right| - h - 1\right) \cdot d_1 \cdot c_{p_0,r_k} + c_{p_0,r_k} \cdot \sum_{j=2}^{\left|P_{||}\right|}\left(\frac{\left|P_{||}\right|}{j} \cdot a_j\right) \\ t_{seq}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} &\leq t_{pll}\left(p_h, r_k\right)_{c_{p_i,r_k}=c_{p_{i+1},r_k}} \end{aligned} \tag{4.16}$$

Since the lower bound of t_{pll} is greater or equal t_{seq}, Theorem 2 directly follows. □

Conclusion 2. *Intuitively explained, since relative delays for parallel requests constantly increase, cf. Equations 4.1, 4.2, the interference-delay for the parallel overlap is always greater or equal than the interference-delay for sequential issued requests. Therefore, t_{pll} constitutes the worst-case overlap for single-channel resources.*

Multi-Channel Resources

As described in Section 4.1.1 a resource may implement multiple channels to handle concurrent requests in parallel. In the following, the impact on the interference-delay computation is examined.

Definition 11 (Number of Resource Channels)**.** The number of channels for a resource r_k is defined as n_{r_k}. Thereby, n_{r_k} also describes the number of concurrent requests, that can be handled with the same access latency. ■

While n_{r_k} concurrent requests can be handled with the same latency, the latency increases similarly to single-channel resources, once more than n_{r_k} requests are issued concurrently. That is, for every $(i \cdot n_{r_k} + 1)$'th request the latency increases, where $i \cdot n_{r_k}$ represents an integral multiple of n_{r_k}. For instance, a resource with $n_{r_k} = 3$ channels can process three parallel requests with the same delay $d_1 = d_2 = d_3$, only if at least four request are issued in parallel the delay increases. In the case of $n_{r_k} = 3$ it follows, that $d_4 > d_3$ while $d_4 = d_5 = d_6$ and so forth. In general, $d_i = d_{i+1}$ as long as $\left\lceil \frac{i}{n_{r_k}} \right\rceil = \left\lceil \frac{i+1}{n_{r_k}} \right\rceil$.

Definition 12 (Multi-channel Access Lanteny)**.** In accordance to Definition 7 for single-channel resources, the access latency for multi-channel resources is defined as:

$$d_{i+1} = \frac{\left\lceil \frac{i+1}{n_{r_k}} \right\rceil}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \cdot d_i + a_{i+1}, \qquad \begin{array}{l} a_{i+1} = \begin{cases} 0 & \forall i, \left\lceil \frac{i}{n_{r_k}} \right\rceil = \left\lceil \frac{i+1}{n_{r_k}} \right\rceil \\ \geq 0 & \text{else} \end{cases}, \\ i \in \mathbb{N}, i = 1, .., \left|P_{||}\right|, \\ n_{r_k} \in \mathbb{N}, n_{r_k} \geq 2 \end{array} \tag{4.17}$$

■

Consequently, relative delays constantly increase with a rising number of requests if all channels are saturated, cf. Equation 4.18.

$$\frac{d_{i+1}}{\left\lceil \frac{i+1}{n_{r_k}} \right\rceil} \geq \frac{d_i}{\left\lceil \frac{i}{n_{r_k}} \right\rceil}, \qquad \begin{array}{l} i \in \mathbb{N}, i = 1, .., \left|P_{||}\right|, \\ n_{r_k} \in \mathbb{N}, n_{r_k} \geq 2 \end{array} \tag{4.18}$$

As a consequence of Definition 12, the access latencies of requests can be hidden, as long as $\left\lceil \frac{i}{n_{r_k}} \right\rceil = \left\lceil \frac{i+1}{n_{r_k}} \right\rceil$, i.e. additional requests can be processed without increasing the overall execution time. For that reason the parallel overlap scenario, as shown in Figure 4.8, does not safely bound the worst-case for $n_{r_k} > 1$ channels. Furthermore, the worst-case overlap does not solely depend on the access latencies d_i. Instead, also the resource capacities c_{p_i,r_k} have to be considered, since they impact the amount of requests, whose latencies might be hidden. Since c_{p_i,r_k} depends on the actual process characteristics it is not possible to statically define a particular overlap scenario, that upper-bounds the worst-case for an arbitrary combination of d_i, c_{p_i,r_k} and $P_{||}$. Nevertheless, similar to single-channel resources, partial overlap is bounded by the sequential and parallel overlap scenarios, for the same reasoning as for Theorem 1. Thus, it does not need to be considered separately.

In the following, first the effect of hidden latencies on the interference-delay is proven, while afterwards the general case is considered to formally prove the dependency of the worst-case interference-delay from the relation between access latencies and resource capacities.

Theorem 3. *For multi-channel resources, the worst-case interference-delay is upper-bounded by the interference-delay for sequential overlap if the number of concurrent requesters is equal to the number of channels, i.e.* $\left|P_{||}\right| = n_{r_k}$.

Proof. To prove Theorem 3, an arbitrary pair of access latencies $d_i, d_{i+n_{r_k}}$ is considered. Further $P_{||} = n_{r_k}$ is assumed. Following Definition 6, i is related to the number of processes. Accordingly, for $P_{||} = n_{r_k}$ the latencies $d_i, d_{i+n_{r_k}}$ are related to the processes p_{i-1} to $p_{i-1+n_{r_k}-1}$. The interference-delays t_{pll} and t_{seq} are calculated with Equations 4.9 and 4.8, respectively.

According to Definition 1, a PE and likewise a process can only issue a single request at a time. Thus no more than $\left|P_{||}\right|$ requests will be issued in parallel. Since $\left|P_{||}\right| \leq n_{r_k}$, all requests can be handled in parallel, i.e. with the same latency d_i. This simplifies Equations 4.9 and 4.8 as follows:

$$t_{seq} = d_i \cdot \sum_{j=i-1}^{i-1+n_{r_k}-1} c_{p_j,r_k} \tag{4.19}$$

$$\begin{aligned} t_{pll} &= d_i \cdot c_{p_{i-1},r_k} + d_i \cdot \sum_{j=i}^{i-1+n_{r_k}-1} \left(c_{p_j,r_k} - c_{p_{j-1},r_k}\right) \\ &= d_i \cdot \sum_{j=i-1}^{i-1+n_{r_k}-1} c_{p_j,res_k} - d_i \cdot \sum_{j=i}^{i-1+n_{r_k}-1} c_{p_{j-1},r_k} \end{aligned} \tag{4.20}$$

Hence, it follows:

$$\begin{aligned} t_{seq} &\stackrel{!}{=} t_{pll} \\ d_i \cdot \sum_{j=i-1}^{i-1+n_{r_k}-1} c_{p_j,r_k} &\stackrel{!}{=} d_i \cdot \sum_{j=i-1}^{i-1+n_{r_k}-1} c_{p_j,r_k} - d_i \cdot \sum_{j=i}^{i-1+n_{r_k}-1} c_{p_{j-1},r_k} \\ d_i \cdot \sum_{j=i}^{i-1+n_{r_k}-1} c_{p_{j-1},r_k} &\stackrel{!}{=} 0 \end{aligned} \tag{4.21}$$

Since $c_{p_l,r_k} \geq 0$, cf. Definition 3, it follows, that:

$$t_{seq} \geq t_{pll} \tag{4.22}$$

Therefore, Theorem 3 is proven. □

Conclusion 3. *The proof of Theorem 3 shows, that concurrent requests to a multi-channel resource do not increase the interference-delay for parallel overlap, if the number of concurrent requests is not larger than the number of channels. This illustrates the impact of hidden latencies. While Theorem 3 only covers the case of $P_{||} = n_{r_k}$, it is also true for $P_{||} \leq n_{r_k}$, since the same rationale applies. Thus, the interference-delay for sequential overlap t_{seq} upper-bounds the worst-case for such a configuration.*

Theorem 4. *If the number of concurrent requests to a multi-channel resource is defined as $\left|P_{||}\right| = n_{r_k} + 1$, the worst-case interference-delay depends on the relation of access latencies and the resource capacities of the processes.*

Proof. To prove Theorem 4, again an arbitrary pair of access latencies $d_i, d_{i+n_{r_k}}$ is considered. Since $P_{||} = n_{r_k} + 1$, the processes p_{i-1} to $p_{i-1+n_{r_k}}$ are considered.

According to Definition 12, $d_i = d_{i+1} = .. = d_{i+n_{r_k}-1}$. This allows for a similar simplification of t_{pll} (Equation 4.8) as applied for the proof of Theorem 3:

$$\begin{aligned} t_{pll} &= d_{i+n_{r_k}} \cdot c_{p_{i-1},r_k} + d_i \cdot \sum_{j=i}^{i-1+n_{r_k}} \left(c_{p_j,r_k} - c_{p_{j-1},r_k}\right) \\ &= d_{i+n_{r_k}} \cdot c_{p_{i-1},r_k} + d_i \cdot \left(c_{p_{i-1+n_{r_k}},r_k} - c_{p_{i-1},r_k}\right) \end{aligned} \tag{4.23}$$

Further, based on Definition 12, $d_{i+n_{r_k}}$ can be expressed as function of d_i:

$$d_{i+n_{r_k}} = \frac{\left\lceil \frac{i+n_{r_k}}{n_{r_k}} \right\rceil}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \cdot d_i + a_{i+n_{r_k}} = \left(\frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} + 1 \right) d_i + a_{i+n_{r_k}} \tag{4.24}$$

Applying Equation 4.24 to the simplified equation for t_{pll} (Equation 4.23) yields Equation 4.25.

$$t_{pll} = \left[\left(\frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} + 1 \right) \cdot d_i + a_{i+n_{r_k}} \right] \cdot c_{p_{i-1},r_k} + d_i \cdot \left(c_{p_{i-1+n_{r_k}},r_k} - c_{p_{i-1},r_k}\right) \tag{4.25}$$

Based on Equation 4.9 for t_{seq} and the adapted Equation 4.25 for t_{pll} it follows, that:

$$
\begin{aligned}
t_{seq} &\stackrel{!}{=} t_{pll} \\
d_i \cdot \sum_{j=i-1}^{i-1+n_{r_k}} c_{p_j,r_k} &\stackrel{!}{=} \left[\left(\frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} + 1 \right) \cdot d_i + a_{i+n_{r_k}} \right] \cdot c_{p_{i-1},r_k} + d_i \cdot \left(c_{p_{i-1+n_{r_k}},r_k} - c_{p_{i-1},r_k} \right) \\
a_{i+n_{r_k}} \cdot c_{p_{i-1},r_k} &\stackrel{!}{=} d_i \cdot \left[\sum_{j=i-1}^{i-1+n_{r_k}} c_{p_j,r_k} - c_{p_{i-1+n_{r_k}},r_k} + c_{p_{i-1},r_k} - \left(\frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} + 1 \right) \cdot c_{p_{i-1},r_k} \right] \\
a_{i+n_{r_k}} &\stackrel{!}{=} d_i \cdot \left[\sum_{j=i-1}^{i-1+n_{r_k}-1} \left(\frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} \right) - \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \right]
\end{aligned} \tag{4.26}
$$

As result $t_{seq} \leq t_{pll}$ if:

$$
a_{i+n_{r_k}} \geq d_i \cdot \left[\sum_{j=i-1}^{i-1+n_{r_k}-1} \left(\frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} \right) - \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \right] \tag{4.27}
$$

Similarly, $t_{seq} \geq t_{pll}$ if:

$$
a_{i+n_{r_k}} \leq d_i \cdot \left[\sum_{j=i-1}^{i-1+n_{r_k}-1} \left(\frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} \right) - \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \right] \tag{4.28}
$$

Equations 4.27 and 4.28 describe the relation between the arbitration delay for $d_{i+n_{r_k}}$ and d_i, depending on the particular resource capacities c_{p_i,r_k}, such that either $t_{seq} \leq t_{pll}$ or $t_{seq} \geq t_{pll}$. Hence Theorem 4 has been proven. □

Conclusion 4. *From Theorem 4 it can be concluded, that the worst-case interference-delay for multi-channel resources depends on the actual relation of access latencies and resource capacities. Hence, it is not possible to statically define a overlap scenario, that upper-bounds the worst-case. The same rationale applies for $P_{||} \geq n_{r_k}$.*

Theorem 5. *The worst-case interference-delay for multi-channel resources is, upper-bounded by the sequential interference-delay t_{seq}, i.e. $t_{seq}(p_h, r_k) \geq t_{pll}(p_h, r_k)$.*

Condition 3. The arbitration delay $a_{i+n_{r_k}}$ for a request to a shared resource is always significantly smaller than the actual resource access latency d_i, i.e. $a_{i+n_{r_k}} \ll d_i$. ♦

Condition 3 is considered to be valid, since d_i includes the waiting time due to concurrent quests, which also includes arbitration delays, cf. Definition 12.

Proof. To prove Theorem 5, Equation 4.26 is further examined.

For $c_i \leq c_{i+1}$, cf. Definition 10, the following applies to Equation 4.26:

$$
\text{for } n_{r_k} = 1: \quad \sum_{j=i-1}^{i-1+n_{r_k}-1} \frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} = 1 \tag{4.29}
$$

$$
\text{for } n_{r_k} > 1: \quad \sum_{j=i-1}^{i-1+n_{r_k}-1} \frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} \geq 2 \tag{4.30}
$$

$$
\text{for } i > n_{r_k}: \quad 0 \leq \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \leq 1 \tag{4.31}
$$

$$
\text{for } i \leq n_{r_k}: \quad \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} = 1 \tag{4.32}
$$

To upper-bound the right part of Equation 4.26 for multi-channel resources, i.e. $n_{r_k} > 1$, Equations 4.30 and 4.32 are applied:

$$x = \sum_{j=i-1}^{i-1+n_{r_k}-1} \left(\frac{c_{p_j,r_k}}{c_{p_{i-1},r_k}} \right) - \frac{1}{\left\lceil \frac{i}{n_{r_k}} \right\rceil} \geq 1 \tag{4.33}$$

$$\rightarrow t_{seq} \leq t_{pll} \text{ if: } \quad a_{i+n_{r_k}} \geq x \cdot d_i \tag{4.34}$$

$$\rightarrow t_{seq} \geq t_{pll} \text{ if: } \quad a_{i+n_{r_k}} \leq x \cdot d_i \tag{4.35}$$

Equation 4.34 contradicts Condition 3, thus Equation 4.35 applies, which proves Theorem 5. □

Conclusion 5. *It can be concluded, that due to the relation between arbitration delay and actual resource access latency, the worst-case interference-delay for multi-channel resources is bounded by the sequential interference-delay. However, if Condition 3 is shown to contradict a particular target architecture, Equations 4.27 and 4.28 have to be used in order to construct the actual worst-case overlap, based on the given access latencies and process resource capacities.*

4.3.3. Interference-sensitive WCET Computation

Finally, to compute the isWCET bound of a process p_h, its core-local WCET bound t_{s,p_h} and the respective interference-delays $t_{int}(p_h, r_k)$ for all resources r_k are combined according to Equation 4.36. The interference-delay $t_{int}(p_h, r_k)$ is computed as described in Section 4.3.2. That is, for single-channel resources Equation 4.8 is applied, while for multi-channel resource Equation 4.9 is used.

$$t_{is}(p_h) = t_{s,p_h} + \sum_{k=0}^{|R|-1} t_{int}(p_h, r_k) \tag{4.36}$$

4.4. Quality of Service Monitoring Extension

As described in Section 2.4.2, core-local timing bounds naturally overestimate the actual WCET. Likewise, this is true for the WCRA bounds, since the same basic techniques and requirements apply. While a certain amount of overestimation is reasonable and even required in order to fulfil the safety requirements, increased overestimation leads to allocated but unused resources. The amount of idle resources for typical execution behaviours is further increased due to the difference between average-case and worst-case behaviour, cf. Figure 2.6. Moreover, it is assumed, that the inter-process interference drastically increases the difference between average-case and worst-case due to the high variations in access latencies. They are supposed to cause a non-negligible amount of idle resources. For example, the isWCET analysis accounts a much higher amount of resource accesses with interference as they are likely to occur during normal execution. Considering a worst-case access latency increase of roughly a factor of two per additional core illustrates the impact of overestimated inter-process interference for an increasing number of cores.

The combination of resource usage monitoring and isWCET analysis, as described in Sections 4.2 and 4.3, yields a fully static approach. The monitoring is configured with the WCRA bounds, which are determined offline. If a process exhausts its resource capacity it is suspended and might only be re-activated if all remaining processes have finished before the end of the current process frame. Consequently, it is not possible to utilise the described idle resources. To overcome this limitation a QoS extension of the basic monitoring is described in this section. The extension is based on the high likelihood for an application to experience considerably lower access latencies

for most of their resource accesses than considered during isWCET analysis. Hence processes will often finish far ahead of the computed end of the process frame. The proposed QoS extension shall enable the system to utilise these otherwise unused execution time and resource requests.

The basic idea is to modify the suspension action. Once a process p_i finishes or reaches its resource capacity, the remaining WCET bounds, i.e. the maximum execution time that is still required to finish, of all active processes are calculated. Once these calculation indicates unused time towards the end of the process frame, additional resources can be granted to p_i. To compute the remaining WCET bounds, the offline isWCET bound calculation is used at runtime. In order to account for the progress of the processes, their resource capacity requirements need to be updated. Afterwards, the maximum over the re-calculated WCET bounds is compared against the remaining time of the process frame. If the comparison reveals unused time, the resource requests (c_{ex}), that can be issued within this idle time, can be assigned to p_i, without violating the deadline guarantees of any process. If, on the other hand no time is left, p_i is suspended similarly to the basic suspension routine.

In the following two sections, the computation of a capacity-extension and foreseen runtime effects are discussed. Afterwards characteristics of potential target applications are analysed.

4.4.1. Computation of Capacity Extensions

The foundation for the capacity extension c_{ex} is Equation 4.36. It is used to compute the remaining WCET bounds based on the actual progress of in-parallel scheduled processes and the absence of interference by already finished or suspended processes. In the following, the process which triggered the suspension routine and thus the computation of the capacity extension, is called the process under investigation. The core-local timing bounds t_{s,p_i}, the initial resource capacities c_{p_i,r_k} per process and resource, as well as the resource access delays d_i are statically known. The elapsed execution time cannot be safely divided into instruction execution and shared resource accesses. Hence, it has to be assumed, that each process still requires its whole core-local WCET bound. Contrarily, the elapsed resource capacities can be directly accounted as resource usage, i.e. only the outstanding capacities need to be considered for the computation of the remaining WCET bounds. Therefor, the remaining resource capacity for each of the active processes has to be updated. The update can either be performed periodically, e.g. timer-triggered or if an update is required by the process under investigation. Based on the updated c_{p_i,r_k}, Equation 4.36 is applied for each process. Since already suspended processes have a resource requirement of $c_{p_i,r_k} = 0$, their interference is automatically disregarded. Depending on the implemented update method, the computed WCET bounds include additional overestimation. That is, a triggered update can rely on a precise value, while values from a periodic update might not reflect the current resource requirements of the processes. Hence, in the latter case a remaining capacity, which is higher than the actual demand, is used to compute the WCET bound. While the overestimation is not an issue for the safety of the approach, it can reduce the overall capacity extension.

The final capacity extension c_{ex} is computed using Equation 4.37.

$$c_{ex} = \frac{t_{pf} - t_{el} - t_{ovh} - \max\limits_{i=0}^{|P_{||}|-1} \left(t_{is}\left(p_i\right)\right)}{d_h} \tag{4.37}$$

The potential idle time within a process frame is determined in the numerator. It considers the already elapsed time of the process frame t_{el}, the length of the process frame t_{pf}, the maximum remaining execution time and the overhead for executing the suspension routine t_{ovh}. The length of the process frame is determined offline, as the maximum isWCET bound of in-parallel scheduled

processes. Hence, it is known at runtime. The elapsed execution time t_{el} can be determined based on the process starting time and the time when the suspension routine was triggered. The remaining WCET bounds are computed based on the updated resource capacities as described above. The overhead t_{ovh} is discussed in Section 4.4.2. The time that can be used for additional accesses is computed by subtracting the elapsed execution time, the overhead and the remaining execution time from the length of the process frame. The resulting value is finally divided by the access delay d_h, to compute the number of resource accesses, that can be issued during the remaining time of the process frame. Thereby, d_h describes the maximum access delay for h processes, with h being the number of remaining processes plus 1 for the process under investigation.

Figure 4.9 shows an exemplary case with $P_{||} = 4$ processes p_0 to p_3, their respective deadlines (dashed, red lines) and resource capacities c_{p_i,r_k} (horizontal, dot-dashed, purple lines) for the shared resource r_{NoC}. The dotted, green line depicts the suspension point of p_3 when it reaches its resource capacity. Upon this point, p_3 acquires the unused resource capacities of p_0, p_1 and p_2 to compute c_{ex}. The individually remaining WCET bounds are shown as continuous, blue lines. The maximum over those values is constituted by p_0 (dot-dashed, pink line). The unused time can thus be expressed as $c_{ex} \cdot d_h$, with $h = 4$.

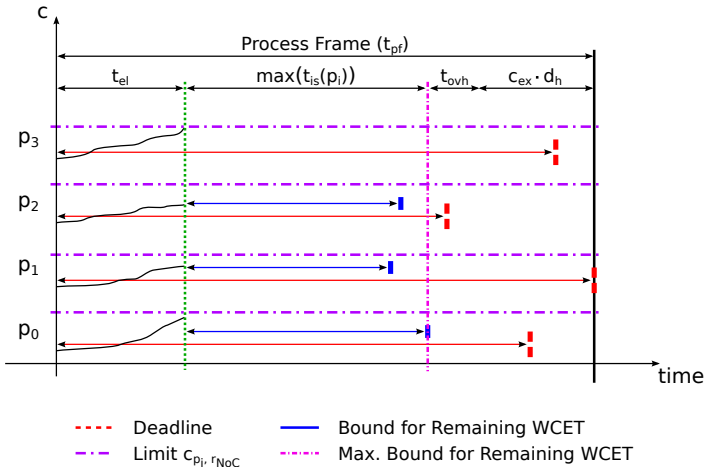


Figure 4.9.: *Capacity-extension example for $P_{||} = 4$ processes and their deadlines (dashed, red lines), resource capacities $c_{p_i,r_{NoC}}$ (horizontal, dot-dashed, purple lines), remaining WCET bounds (continuous, blue lines) and the suspension point (dotted, green line).*

If a process has been granted a capacity extension it is further treated as any other process. Since c_{ex} is computed based on the offline timing bounds and the enforced WCRA bounds, the resource capacity and deadline guarantees of all remaining processes are not violated.

4.4.2. Runtime Effects

The extension of the suspension routine introduces additional sources of interference with in-parallel scheduled processes.

Firstly, the execution of the computation of c_{ex} requires additional resource accesses and execution time, whereat the bounds for WCET and WCRAs of the routine need to be determined. The computation of the isWCET bound and c_{ex} solely depends on the number of shared resources and PEs. Since they are statically known for a given target architecture, the WCET and WCRAs of the computation of the capacity extension can be bounded offline. The additional execution time only influences the process under investigation. Since the suspension of a process is an exceptional

case, the additional execution time does not need to be incorporated into the WCET bound of the process. But, since the computation uses a portion of the process frame, the WCET bound for calculating the capacity extension has to be considered as overhead. As such, it is part of t_{ovh}. In addition to the execution time overhead, also the issued resource requests, in particular their interference with in-parallel scheduled processes, has to be considered. Based on the determined WCRA bounds of the suspension routine, the overall WCRA bounds of each process are augmented. Accordingly, the additional interference is accounted during the isWCET bound computation. Obviously, this only covers a single execution of the routine. To cover further executions, the computed capacity extension is compared against the WCRA bound of the suspension routine. A process is extended, only if the capacity extension enables at least one additional execution of the suspension routine. Thereby it is ensured, that the additional requests of the next extension do not violate the deadlines of the active processes. If the computed capacity does not allow an additional extension, the process is suspended similarly to the basic routine.

Secondly, the update of the remaining resource capacities causes overhead and, depending on the implemented method, interference with other cores. Except for software instrumentation based approaches, an update request is always externally triggered, i.e. is not part of the application code itself. For such approaches, all side effects due to the redirection of the control flow, for instance on the PE pipeline and cache content, have to be considered additionally. With state of the art timing analysis techniques it is not possible to provide reasonable bounds for the behaviour of a process, that can be preempted at arbitrary program points. However, the effects due to the communication are similar to those of a preemption. Thus, techniques to analyse the effects of preemptions can be applied, e.g. [Staschulat and Ernst (2004), Ramaprasad and Mueller (2006)]. If the exact point in time of a preemption is not known, the detailed effects on the PE and application state cannot be analysed. Instead, worst-case assumptions, such as fully invalidated caches or an empty pipeline, are considered. The overheads of the preemption are determined based on the preempting code. Thus, in the present case, the core-local bounds for WCET and WCRAs of the capacity update routine need to be determined. Finally, the overheads are added with a constant factor to the bounds of the application. The factor depends on the maximum number of preemptions. As described, the communication can either be timer- or PE-triggered. In case of periodically triggered updates, the maximum number of preemptions is determined by the WCET bound of an application divided by the length of the update interval. The case of a PE-triggered update can be handled similarly, by applying a minimum interval between two update requests. If a PE requires an update earlier, it either has to wait until the end of the interval or use the available older data. While applying older data is safe, it might underestimate the capacity extension, since applications probably made some progress since the last update. Overheads for the communication and potential waiting times for the process under investigation can be measured at runtime and accounted as part of t_{ovh}.

4.4.3. Target Applications

The described capacity extension is build upon the assumption, that the average-case execution time greatly deviates from the WCET bound of a process. Beneath that, the extension inherently assumes, that applications can benefit from additional execution time and resource requests. It can easily be understood that this is not necessarily true for all applications.

Firstly, the application characteristics can indicate potential benefits for applications. For instance considering a control loop application, that has to deliver steering parameters in a fixed interval to other applications. A more frequent computation of the parameters will not increase the quality of the system, since the depending applications will not be able to further process

the data. On the other hand, an application, which implements some kind of optimisation-based algorithm might increase the level of detail of the optimisation if it gets more processing time and resource requests. Examples for the latter case can be flight path computation to optimise the trajectory of plane to reduce the fuel consumption, and display applications that can offer higher frame rates, resolutions and improved response times, which increases the overall user experience.

Secondly, the applied timing analysis techniques can enable benefits. In particular, an extension can be useful for applications whose bounds for WCET and WCRAs are determined via measurement or hybrid approaches, cf. Section 2.4. Measurement-based approaches in general cannot be assumed to acquire safe upper bounds. In the context of the capacity extension concept, the so determined bounds can be used as basis, which allows a process to finish in most of the cases. In this case, the capacity extension increases the probability for such an application to still finish under worst-case conditions, since in-parallel scheduled applications will probably not experience worst-case conditions at the same time . This can similarly be used to reduce the isWCET bound of processes. The initial resource capacity is lowered to reduce the isWCET bound, while during most of the executions, the total amount of resource requests will be much higher due to the capacity extension.

Considering the different application scenarios it is not meaningful to use capacity extensions for all applications. Instead, the *capacity-extendible* property is introduced. This property is assigned to each process and determines whether this process is enabled for capacity extensions. For processes whose capacity-extendible property is not set the basic suspension routine is executed. As such, the suspension routine has to check if the property is set for a given process.

4.5. Integration of Explicitly Shared Resources

So far the monitoring and isWCET analysis handle the interference between PEs, caused by the concurrent usage of implicitly shared resources. This section shall additionally cover the interference, that can be introduced by explicitly shared resources. Explicitly shared resources are divided into master and slave devices, according to their abilities to access implicitly shared resources, cf. Figure 4.1. Slave devices only transfer data once they are triggered by software, i.e. the respective device driver. On the other hand, master I/O devices are able to initiate data transfers on their own. From a system perspective, such DMA transfers are comparable to resource accesses by PEs.

In the following the integration into the monitoring as well as the isWCET analysis are discussed.

4.5.1. Monitoring Integration

The monitoring integration of explicitly shared resources covers the monitoring of implicitly shared resource requests of I/O devices and the different means, that are available in the resources themselves, within the SoC and on software level. Therefore, the monitoring hardly depends on the characteristics and features of the individual device. This concerns how data transfers from the device are controlled - software controlled, as for slave and some DMA-I/O devices, or independently via externally triggered DMA transfers. Also the available features in the devices and the SoC architecture affect the monitoring.

Software-controlled Devices

Usually, applications do not interact directly with I/O devices, rather then through a dedicated device driver. The device driver has full control of the device configuration. For slave devices and some DMA-I/O devices, the driver also controls the amount of data it transfers. Every time data needs to be transfered from the device to the system or vice versa, this is explicitly configured by the driver. For that purpose, the driver is able to record all transfers and compare the amount of data against the resource capacity of the device. If an exhaustion of the capacity has been detected, the driver does not initiate the transfer and thereby avoid further interference on the respective implicitly shared resource.

This method of software controlled monitoring is generally applicable to all slave devices, but also to some DMA-I/O devices. In particular DMA-I/O devices must be able to buffer all data until software triggers a transfer, i.e. they do not initiate DMA transactions on their own. To prevent data loss it has to be ensured, that the buffers of the DMA-I/O devices are large enough to store the data till the next transfer.

Externally Triggered DMA Devices

If it is not feasible to use fully software-based monitoring, e.g. because devices are required to initiate DMA transfers at any time, different means to monitor the device transfers are required. Figure 4.10 shows three approaches for hardware-supported monitoring of DMA-I/O devices.

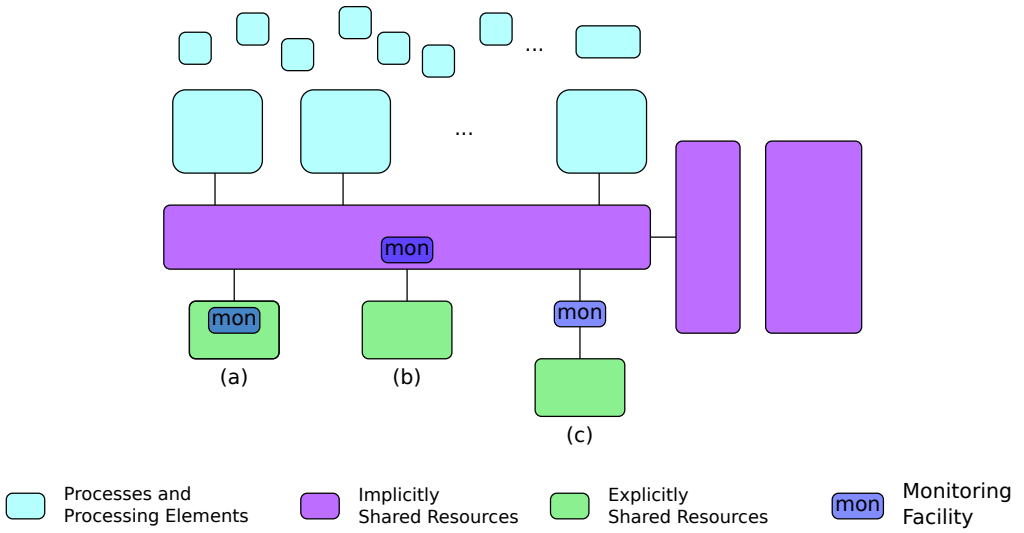


Figure 4.10.: *DMA-I/O-device monitoring facility locations: (a) device-built-in monitor (b) SoC and implicitly shared resource monitor (c) I/O-bridge monitor.*

(a) Device-built-in Monitor: Device-built-in monitoring facilities are comparable to Performance Monitor Counters (PMCs) within the PEs. In order to be useful, they are required to be able to count the chosen resource parameter(s). Further the ability to compare the current values against the given WCRA bounds and trigger a notification to software once the resource capacity is exhausted, is desirable. If the latter feature is not supported, a periodic polling of the counter in software is required. In this case the inaccuracy, i.e. the exceedance of the resource capacity without immediate reaction, has to be taken into account when assigning the WCRA bounds.

(b) SoC and Implicitly Shared Resource Monitor: If the DMA-I/O device does not provide sufficient monitoring facilities, similar features in the SoC and the targeted implicitly shared resource can be utilised. As described in Section 4.2, modern SoC architectures implement extensive debugging and monitoring features to overcome the limited visibility of the chip behaviour due to the high integration level. The functionality is similar to device-built-in monitoring counters, hence the same implementation applies. However, since those SoC monitoring facilities are shared among all devices within the system, it has to be ensured that enough instances are available to cover all devices.

(c) I/O-Bridge Monitor: When neither of the monitoring facilities are available, so-called I/O-bridges can be used to extend the system functionalities. The bridge is implemented as an interface between the I/O device and the SoC, implementing the monitoring functionalities. The I/O-bridge proposed in [Pellizzoni and Caccamo (2007), Pellizzoni et al. (2008)] is implemented as a PCI/PCIe debug extender card. The card connects the target system and the PCI/PCIe device. While such an implementation is valid for interfaces, that provide SoC external connectors, it is not for interfaces, that are integrated into the SoC. If such integrated devices also cannot be covered by the means of (a) or (b), they cannot be integrated within the proposed approach. As an alternative, their functionality can be replaced by an external plug-in card in combination with one of the proposed integration methods.

For a particular device the choice of the appropriate monitoring method depends on the device, the SoC features and the system requirements with respect to the performance of the device and the combination with other devices. For instance, if high data rates are required, it is advisable to allow device triggered DMA transactions; if the monitoring facilities of the SoC are already assigned to other devices, a different approach has to be used or the data transfers of the device need to be scheduled at a different process frame, if possible.

4.5.2. Interference-sensitive WCET Analysis Integration

When PEs and explicitly shared resources share the same implicitly shared resource it is required to have common partitioning parameter(s), in order to be able to analyse their impact on each other. Since the communality is the implicitly shared resource, the parameter(s) should be chosen depending on that resource. For example, when sharing the system interconnect, its bandwidth, i.e. the number of requests in a certain period, would be an adequate basis. Correspondingly, the requests of PEs and DMA-I/O devices need to be translated into NoC transactions. For instance, if the DMA-I/O device transfers packets of a given size s while each NoC transaction consists of $n \cdot s$ packets, n packets of the device are grouped to a single NoC packet. This transformation from device to NoC packets also covers different DMA modi, such as single-packet and block transfers.

In order to apply the isWCET analysis, an upper bound on the total amount of requests of a DMA-I/O device within a process frame has to be determined. Once the translation from the device specific requests to the implicitly shared resource transactions and the upper bound are known, the isWCET analysis can be applied. Since the requests of PEs and DMA-I/O devices are translated to the same basis they are commonly considered as requests. In accordance to the example for PEs in Figure 4.8, the interference-delay analysis of in-parallel scheduled processes is augmented by the requests of DMA-I/O devices, which transfer data within the same process frame. The set of DMA-I/O devices whose requests are scheduled to the same process frame, is further referred to as $IO_{||}$. In order to extend the computation of the interference-delay, the sets of in-parallel scheduled processes $P_{||}$ and explicitly shared resources $IO_{||}$ are combined.

Definition 13 (Masters Devices)**.** The set of master devices MA is defined as the combination of the set of processes P and the set of explicitly shared resource IO. Based on this, the set of in-parallel scheduled master devices $MA_{||}$ is defined as:

$$MA_{||} = \{ma_m\}, \quad \begin{array}{l} \forall m \in \left[0, \left|P_{||}\right| + \left|IO_{||}\right| - 1\right), \\ ma_m \in P_{||} \vee ma_m \in IO_{||} \end{array} \tag{4.38}$$

■

In order to compute the interference-delay, Equations 4.8 and 4.9 are extended as follows:

$$t_{seq}\left(ma_h, r_k\right) = d_1 \cdot \sum_{i=0}^{h} c_{ma_i, r_k} \tag{4.39}$$

$$t_{pll}\left(ma_h, r_k\right) = d_{\left|MA_{||}\right|} \cdot c_{ma_0, r_k} + \sum_{i=1}^{h} \left(d_{\left|MA_{||}\right|-i} \cdot \left(c_{ma_i, r_k} - c_{ma_{i-1}, r_k}\right)\right) \tag{4.40}$$

As for the computation of the interference-delay for PEs, it is required that the capacities c_{ma_i, r_k} are sorted in ascending order. Equation 4.36 to compute the final isWCET bound does not need to be adapted, instead the adapted Equations 4.39 and 4.40 have to be used to compute the interference-delay. Since an I/O device does not provide a core-local WCET bound, it is assumed as zero. Finally, Equation 4.36 can be applied, whereat p_h has be replaced by ma_h. An example for the computation of the parallel interference-delay is shown in Figure 4.11.

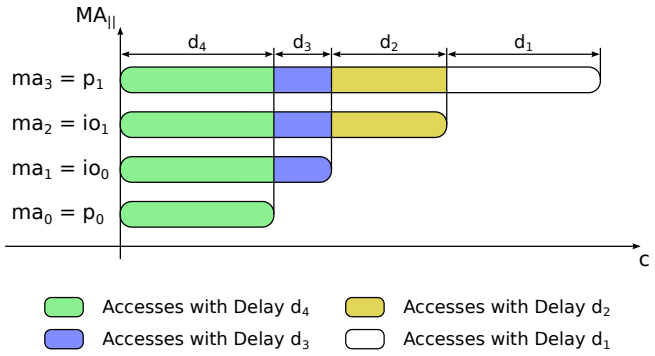


Figure 4.11.: *Interference-delay computation example, for processes p_0, p_1 and DMA-I/O devices io_0, io_1, showing their capacities c and the respectively applied latencies d_i.*

4.6. Process Integration

To integrate multiple applications on a target platform in the context of the presented monitoring and analysis approach, a higher-level workflow is required. Such a workflow for the software development is described in this section. It covers the basic workflow, the integration of the monitoring and analysis approaches, as well as strategies to modify and optimise the computed isWCET bounds towards superior system requirements.

4.6.1. Software Development Workflow

The workflow is depicted in Figure 4.12. It is focused on the *Software Development* but also shows relevant parts of the *System Development* and *Hardware Development*, in approximation of the

avionics guideline documents as shown in Figure 2.1. According to industry standards as DO-178 [RTCA (2012)] and ISO 26262 [ISO (2011)], development processes are requirements-based. The first step in the software development is the software design. It is used to derive the software requirements from the system functions and requirements, which were defined during system development. Software requirements for example include desired functions and applications, data dependencies, as well as timing and resource constraints. Initial timing requirements are usually assigned based on known constraints, e.g. on the execution frequencies and execution times, but also based on experience from previous systems. During software design, the software requirements are translated into the software architecture, which afterwards is implemented during software development. To verify that timing constraints are met, timing analysis is applied. As described in Section 4.3, the proposed timing analysis is split into core-local and interference-delay analysis. The core-local analysis determines bounds for WCET and WCRAs for each application and passes those information to the interference-delay analysis. The interference-delay analysis uses those values to compute the isWCET bounds for in-parallel scheduled applications. Both steps of the timing analysis require architecture parameters, such as a pipeline model if static analysis is applied, and the resource access delays depending on the number of requesters. The architecture parameters are determined within the computing architecture analysis as part of the hardware development. Additionally, the interference-delay analysis requires the set of in-parallel scheduled processes as an input. The initial mapping from processes to PEs and process frames is computed during the scheduling analysis. Unfortunately, the scheduling analysis requires the isWCET bounds of the applications, while the interference-delay analysis requires the process mapping computed during scheduling analysis. To account for this circular dependency, the initial schedule is computed, based on the timing and resource constraints, which are defined within the software architecture. The resulting isWCET bounds can be used within further iterations, to optimise the system schedule and vice versa, until the requirements of the software architecture are fulfilled.

Beneath checking the schedulability of the processes, the scheduling analysis also has to prove, that Equation 4.41 is fulfilled. It ensures, that the resource requirements of in-parallel scheduled process and explicitly shared resources do not overuse the respective resources.

$$c_{r_k} \geq \sum_{i=0}^{|MA_{||}|-1} c_{ma_i,r_k}, \quad \forall r_k \in R \tag{4.41}$$

If it is not possible to define a schedule which fulfills the requirements, feedback edges (dotted arrows) in the workflow allow to either change the implementation or the overall software architecture, e.g. by relaxing the constraints for applications which did not meet the requirements, while restricting other applications whose initial timing and resource contingents are not exhausted. This describes a typical system engineering refinement approach. Once a final iteration through scheduling and interference-delay analysis identifies a valid configuration, its parameters, i.e. the process activation points and their bounds for WCET and WCRAs constitute the software configuration. This also includes the length of each process frame, which is determined by the maximum isWCET bound over the in-parallel scheduled processes and I/O devices. In combination with the software implementation they are used during execution at runtime.

While the isWCET analysis is part of the described workflow, the monitoring is a runtime mechanism. As such it is part of the software architecture. Accordingly, the defined functional and temporal transparency, cf. Section 4.2, provide additional requirements to the software architecture.

As indicated in Figures 2.1 and 4.12, the software development is only one part of the processes to

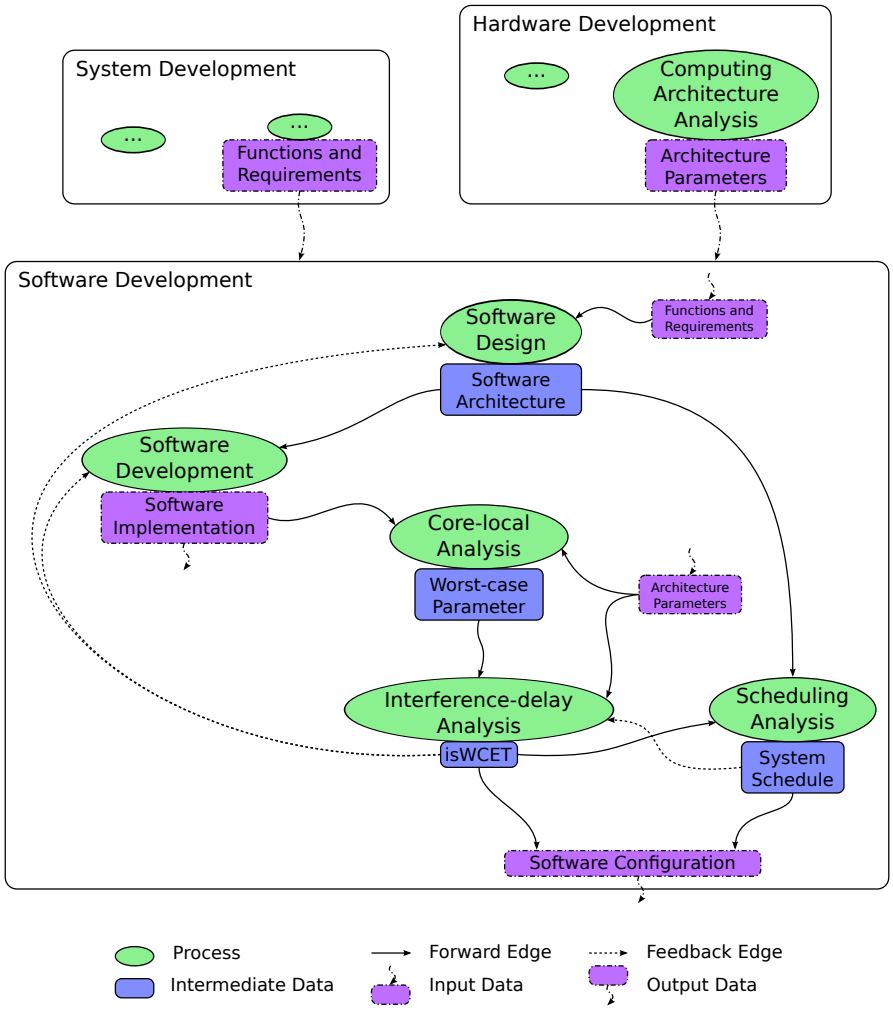


Figure 4.12.: *Exemplary software development workflow with integrated isWCET analysis.*

develop a complete system. As such, the presented workflow shall be understood as an exemplary integration of the described approaches into the system development.

4.6.2. Optimising the Interference-sensitive WCET Bounds

According to the presented workflow, the system design defines initial values for WCET bound and resource capacities based on predefined requirements and experience. The final values are acquired during iterations of the workflow. In general the isWCET bound depends on the WCRA bounds of in-parallel scheduled processes and I/O devices. Hence, to influence the computed timing bounds, the relation between the WCRA bounds of the respective processes has to be modified. This can either be obtained by alternative schedules, i.e. changing the sets of in-parallel scheduled processes and I/O devices or by directly changing the capacities of processes. Modifications to the schedule are already included in the workflow, indicated by the iteration between interference-delay and scheduling analysis. A general heuristics to optimise a schedule is to combine processes with relatively low WCRA bounds with processes with higher WCRA bounds. Further, the WCRA bounds of processes can directly be modified to, thereby influence the isWCET bounds. In the context of mixed-criticality systems the WCRA bounds of a process can be reduced, as long as

the probability of a suspension due to a capacity exhaustion still suits the design assurance level of that process. Additionally, the QoS extension can be considered pro-actively by lowering the WCRA bounds of a process, such that a minimal service can be guaranteed, while relying on the QoS extension to increase the average-case performance during actual execution.

5. Implementation

This chapter describes the implementations for the different aspects of runtime resource usage enforcement and isWCET analysis based on the described functionalities and discussed alternatives in Chapter 4. It is divided into the description of the target architecture and the software layers in Sections 5.1 and 5.2, the resource usage enforcement, cf. Section 5.3 and the timing analysis in Section 5.4.

The implementation is focused on the NoC as the major implicitly shared resource. The concurrent use of other shared resources can be avoided. For instance, to avoid concurrency on the main memory, different memory controllers can be mapped to individual cores while other cores can load data and instructions out of a completely different source. However, the NoC in modern SoCs usually connects all parts of the system, thus it is an integral part of the system which has to be used. As explained in Section 4.2, from a core and application perspective requests to individual parts of the memory hierarchy, i.e. platform caches and main memory, cannot be distinguished. Therefore, the off-core memory resource has been introduced, which abstracts the set of NoC, platform caches and the main memory.

5.1. Target Architecture

The Freescale P4080 [FSL (2012)], a state of the art COTS MPSoC, has been chosen as target architecture. It is the reference platform for the Freescale QorIQ series of processors. The decision towards this platform is based on experiences with platform vendors for previous airplane computer systems as well as on the choices of recent evaluation platforms for future safety-critical systems. As such, the P4080 is commonly used in research and industry to evaluate potential multi-core architectures [Jean et al. (2012)]. It is under investigation in multiple research projects, such as ARAMiS [BMBF (2011)], MUSE [FOKUS (2010)] and RECOMP [ARTEMIS (2010)].

Figure 5.1 shows a block diagram with the main functional blocks of the P4080. It contains the PEs, examples for implicitly and explicitly shared resources and the SoC-level debug facility. The following information are derived from the PowerPC ISA definition [IBM (2010)], e500mc core manual [FSL (2011)] and the P4080 platform manual [FSL (2012)].

e500mc cores: The P4080 implements eight symmetric e500mc, 32*bit* PowerPC cores. The core-local memory hierarchy includes separate L1 instruction and data caches ($32KB$, eight-way set-associative, $64B$ cache lines), a unified L2 cache ($128KB$, eight-way set-associative, $64B$ cache line) as well as separate mini instruction and data caches, termed Data Line Fill Buffer (DLFB) ($320B$, fully-associative, $64B$ cache line) and Instruction Line Fill Buffer (ILFB) ($128B$, fully-associative, fully-associative, $64B$ cache line), respectively. The L1 caches implement the Pseudo Least Recently Used (PLRU) replacement strategy, while the L2 cache can be configured to apply different streaming variants of the PLRU strategy. The elements of the core pipeline and the core-local memory hierarchy are connected via the core interface unit, likewise the core interface unit is connected with the SoC through the Bus Interface Unit (BIU). The core further contains a 2-way superscalar pipeline with out-of-order execution and in-order completion. The pipeline implements simple and complex integer as well as floating

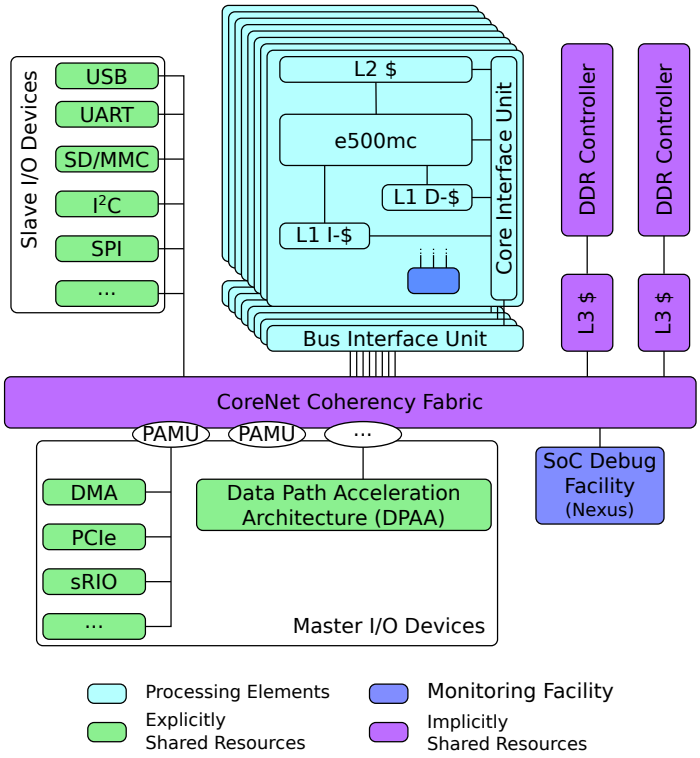


Figure 5.1.: *Freescale P4080 block diagram, cf. [FSL (2011), FSL (2012)].*

point, load/store and branch predictions units. For inter-core communication, the PowerPC architecture defines a so-called doorbell exception mechanism. It can be used to trigger exceptions on other cores to redirect their control flow without an associated data transfer.

Off-core Memory: As stated above, the off-core memory includes the NoC, called CoreNet, the level-3 (L3) platform cache and the main memory. CoreNet is the central interconnect in the P4080, hence most of the device data transfers have to pass through the NoC. Exceptions are dedicated communication channels between parts of the Data Path Acceleration Architecture (DPAA), the network acceleration units. The interconnect also implements the coherency mechanisms. Since, only very little information on the architecture of CoreNet are publicly available, it is considered as a black box NoC. However, it is known, that it can process up to four transactions in parallel. Thus, it could be considered as multi-channel resource. The platform cache and the main memory provide two separate controllers each, while one L3 controller is assigned to one main memory controller. For that purpose, each pair of L3 cache and main memory controller can be considered as individual shared resources with a single-channel. As explained, from a core perspective the actual target of a load/store type instruction cannot be determined in beforehand. That is, each instruction targets main memory, but might be contained in the L3 cache. While the two pairs of L3 and main memory controller can be distinguished via the physical address, it cannot be pre-determined which channel of CoreNet is used for a certain transaction. For that purpose, the off-core memory is considered as two separate resource instances with a single-channel for each of them. It is further referred to as r_{NoC}, i.e. $c_{p_i,r_{NoC}}$ defines a bound for the number off-core memory requests for process p_i.

Explicitly Shared Resources: As a SoC architecture, the P4080 implements a variety of peripheral interfaces. This includes typical slave I/O devices, such as SPI, UART and I^2C, and DMA-I/O interfaces, like PCI/PCIe, sRIO and dedicated DMA controllers. For spatial separation

the P4080 implements units called Peripheral Access Management Unit (PAMU), which are the I/OMMU implementation by Freescale. As such, they ensure the system address space separations also for DMA-I/O devices by checking the target of each DMA transaction against a mapping table. Accordingly, each DMA-I/O device is connected to CoreNet through a PAMU.

Monitoring Facilities: The P4080 implements a SoC-level debug and performance monitoring facility, based on the Nexus standard [IEEE-ISTO (1999)]. This includes shared units for central information gathering and processing as well as local monitoring facilities for the individual units of the system, such as the e500mc cores. Details of the monitoring facility structure and functionalities are described in Section 5.3.

5.2. Software Layers

This section describes the different software layers and their purposes, functionalities and interactions. The general software structure is shown in Figure 5.2. Figure 5.3 illustrates the control flows and interactions between different parts of the software layers.

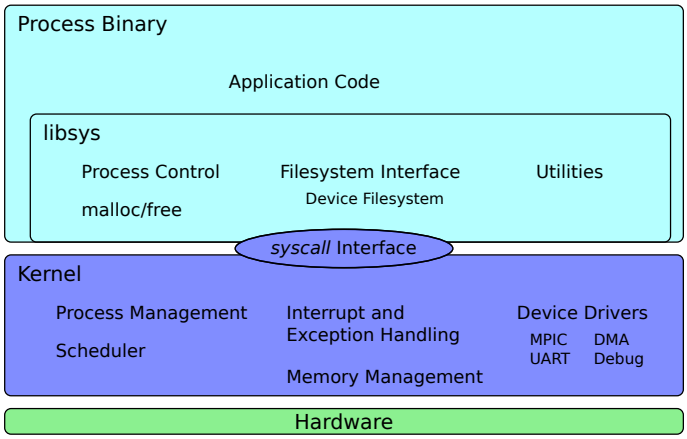


Figure 5.2.: *Software structure and address spaces.*

The software system is based on a bare-metal operating system layer, which in accordance to Section 2.3 is referred to as SK. The SK has been developed as part of this thesis, its main functionalities are the configuration and control of the underlying hardware, process management and the implementation of temporal and spatial partitioning as described in Section 2.3. The SK executes in the highest available privilege level, while the application software runs with reduced privileges in isolated address spaces. As shown in Figure 5.2, the SK implements modules to manage processes, memory allocation and exceptions. It interfaces the hardware via different devices driver. In the current version drivers for the UARTs, the Multi-core Programmable Interrupt Controller (MPIC) and DMA controllers are implemented. On the other hand, the SK provides an interface to application software via System Calls (syscalls). The syscalls are grouped in memory and process management, as well as filesystem calls which interface with the device drivers via the device-filesystem. The syscalls are combined with basic utility functions into the system library (libsys), which is linked with the object files of each application constituting a complete process binary. Likewise, the SK is compiled into a separate binary.

Based on the structure of the software, Figure 5.3 depicts the control flow of the software system and interactions between individual parts.

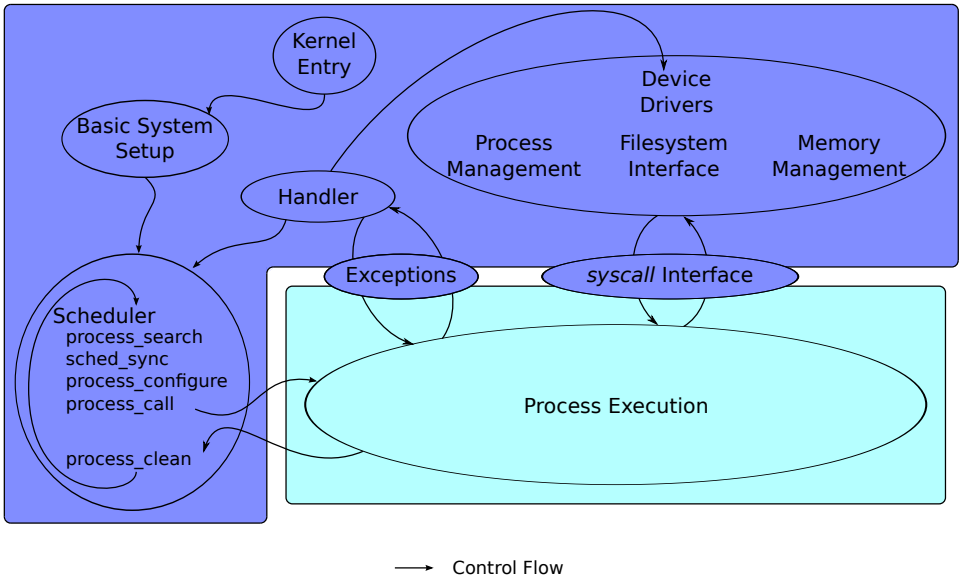


Figure 5.3.: *Interactions and transitions within the software system.*

On reset all but one core of the P4080 are disabled. The remaining core starts executing the SK at its entry point once it has been loaded to memory. Its first task is the basic system setup, which includes the initialisation of the underlying hardware, internal management structures and device drivers. The final step of booting the system is to enable the inactive cores. Once the initialisation is finished all cores execute the scheduling loop, periodically checking for available processes. If a process has been found it is executed once the next scheduling event is triggered. Therefor, the core MMU is configured and the processor context is switched from kernel to user space. A process executes until it finishes its task or the next scheduling event occurs. The scheduling event is a global exception, triggered via a counter in the MPIC and broadcasted to all cores. Thereby, the implementation conforms to the described scheduling scheme in Section 4.1.2. During process execution, the software can use the syscall interface to communicate with the kernel, e.g. to allocate and free memory, create new processes and interact with the hardware. Further, exceptions can redirect the control flow from a process to the kernel. The only sources of external events within the current configuration are exceptions from the cores indicating errors, inter-core communication, i.e. doorbell exceptions, and the UART to signal data data transfers from the host system. Any other sources within the P4080 are disabled through the MPIC. Hence, the SK provides a very deterministic execution environment, especially compared to standard embedded operating systems, which from an application perspective suffer more or less random context switches and software concurrency.

5.3. Runtime Resource Usage Enforcement

Within this section the implementation of the runtime resource usage enforcement is described. While in Section 4.2 software-based and hardware-assisted solutions have been discussed, the implementation is based on the latter approach. Software-based approaches are disregarded due to the static source code and runtime overhead of the instrumentation, which can easily be avoided by hardware-assisted approaches, whose prerequisites are commonly available in modern architectures. For that purpose, first available implementation alternatives in the P4080 are examined

while afterwards the implementation of the basic monitoring as well as of the QoS extension are described. According to the described workflow, cf. Figure 4.12, the resource capacities and core-local WCET bounds are determined via offline timing analysis. Thus, they are available at runtime as part of the system configuration.

5.3.1. Monitoring Alternatives

Core Performance Monitoring Counter

The PowerPC architecture [IBM (2010)] defines so-called PMCs as shown in Figure 5.4. These are counter registers which can be triggered on various events generated within the core pipeline. Once enabled, each counter can be started and stopped automatically, e.g. depending on the current processor privilege level. If the configured event occurs the counter is incremented. Each counter can be configured to trigger an exception in reaction to an overflow and watchpoint events on specific counter values. Further counters can be chained to increase the overall length and thus the range of values.

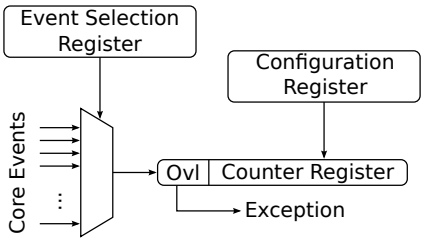


Figure 5.4.: *PowerPC PMC simplified schematic, cf. [FSL (2011)].*

The e500mc cores within P4080 provide four 32bit PMCs, which can be triggered on up to 256 events. Examples for such events are the number of fetched instructions, completed instructions, individual instruction types and cache-related events. With respect to the monitoring of off-core memory requests, the BIU request event is of particular importance. It includes explicit instruction fetches, data load/store operations as well as accesses caused by pre-fetching and cache-related write-backs. That means, as indicated by its name, the BIU request event is triggered on every transaction which crosses the interface between the core and the NoC.

SoC Debug and Performance Monitoring Facility

As mentioned in Section 4.2, modern SoC architectures implement powerful debugging facilities to provide sufficient insights into the actual system behaviour. The debug architecture of the P4080 implements the Nexus standard [IEEE-ISTO (1999)]. It includes an event and a trace path, whereat the event path signals the occurrence of various events, while the trace path handles more exhaustive information, such as timing information. Figure 5.5, shows an overview of the event path. The trace path is not further discussed, since it is not required for the desired monitoring mechanism.

The debug architecture is divided into component-specific and commonly used, cross-component functional blocks. The component-specific blocks apply a pre-selection of events based on the particular target component, e.g. main memory, core or network interface. Hence only a subset of the potential events can be traced during execution. Once any of the monitored events occurs, the component-specific unit forwards it to the cross-component blocks. Accordingly, the cross-component blocks gather all events from within the SoC. Further they are able to apply

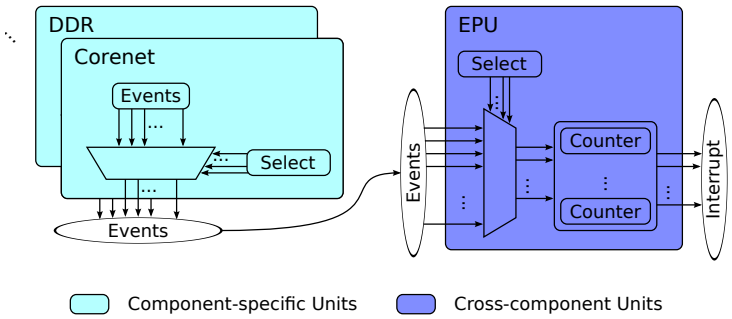


Figure 5.5.: *P4080 debug and performance monitoring architecture event path, cf. [FSL (2012)].*

post-processing and trigger subsequent events, such as interrupts. Beneath others, the cross-component blocks contains the Event Processing Unit (EPU). The EPU can be used to apply post-processing and combination of events, which also includes a counting ability. In combination with the capability to generate interrupts, the same kind of functionality as with the core-local PMCs can be realised.

Out of the available component-specific units, only the CoreNet-specific component can be used to trace accesses from the cores to the off-core memory. While also the main memory controller provides a component-specific block, it is not sufficient to rely on those events, since it cannot trace requests which are processed by the platform cache.

5.3.2. Basic Monitoring

Considering the functionality of the core-local PMCs and the SoC debug architecture both of them can be used to implement the runtime resource usage enforcement. However, while the SoC-level solution could be configured as desired, the implementation revealed architectural limitations. In particular, the CoreNet-specific block does not provide a sufficient number of event selection facilities to apply the monitoring on each of the eight cores, which is essential to provide a safe mechanism. Further, it is not possible to trace all required event types, i.e. read and write requests. On the other hand, each core supplies private PMCs and the possibility to count all off-core memory requests. For that purpose the final monitoring implementation leverages the core-local PMCs as described in the following.

The implementation is based on the described bare-metal SK. The functionality of the PMCs is covered in a device driver. The driver provides the following functionalities:

pm_config(): The PMCs require a global configuration which defines common settings for all the counters within a core. For the present case mainly the exception triggering mechanism needs to be enabled. In order to safely handle occurring exceptions the respective callback functions need to be registered. Listing 5.1 shows the corresponding source code. First the callback for the INT_PM exception, which is the exception associated to the PMCs, is registered to int_hdlr_pm(), afterwards the exceptions are globally enabled.

```
1 void pm_config(){
2     /* register PM exception callback function */
3     int_register(INT_PM, int_hdlr_pm);
4
5     /* enable exceptions for PMCs */
6     reg_write(PMC_GLB_CFG, EXCP_EN);
```

```
7 }
```

Listing 5.1: *Global PMC configuration function.*

pm_cnt_config(): The pm_cnt_conig() function is responsible for configuring a PMC. This comprises the selection of the event of interest, the configuration of the initial counter value and the activation of the counter. Listing 5.2 shows the respective code snippet. The only argument to the function is the desired capacity, i.e. the WCRA bound $c_{p_i,r_{NoC}}$ of the current process. Since the PMCs only trigger exceptions on counter overflows, the initial value of a counter is computed as $0x80000000 - c_{p_i,r_{NoC}}$, whereas $0x80000000$ represents the first value of a 32bit counter which triggers an overflow.

```
1 void pm_cnt_config(unsigned int c_{p_i,r_{NoC}}){
2     /* set initial counter value */
3     reg_write(PMC_VAL, 0x80000000 - c_{p_i,r_{NoC}});
4
5     /* enable counter and select evt */
6     reg_write(PMC_CFG, PMC_EN | EVT_BIU_ACCESS);
7 }
```

Listing 5.2: *Configuration function for a single PMC.*

int_hdlr_pm(): int_hdlr_pm() is the callback routine, which is invoked by the PMC ISR. The ISR is only triggered on the overflow of a PMC, hence int_hdlr_pm() has to implement the suspension routine. As defined in Section 4.2, the suspension routine has to avoid any further accesses of the process to the shared resource until the end of the process frame or till all in-parallel scheduled processes have finished execution. The e500mc cores do not provide means to cut the connection between core and off-core memory. Hence, a process has to be suspended completely. Listing 5.3 shows the respective source code.

```
1 void int_hdlr_pm(){
2     /* suspend current process */
3     process_suspend(core[this].current_process);
4
5     /* switch ISR context to kernel */
6     next_process = kernel;
7 }
```

Listing 5.3: *PMC overflow exception handler.*

First the subroutine process_suspend() is called, which, for evaluation purposes, sets the return value of the process. In practical implementations this subroutine would include the described error handling mechanisms. Afterwards the processor context is modified such, that the ISR does not return to the process, but instead to the kernel. Once the ISR is finished the core continues with the execution of the scheduler loop. As shown in Figure 5.3, once the process context has been cleaned, the core searches for the next process and waits for the next scheduling event to occur. If no suitable process has been found, the wait is executed immediately. This is implemented such, that it puts the core pipeline to a halting state, waiting for an exception. Therefore, it does not issue new resource requests. The current suspension action does not handle outstanding write-backs. That is, the core-local caches still contain instructions and data of the process, which will cause additional write back transactions once they are evicted. In productive systems such outstanding requests either need to be considered during the interference-delay analysis or avoided completely by invalidating the cache content at runtime, i.e. clearing the valid bit of each cache line without updating system memory.

Since the exhaustion of the resource capacity and the following suspension of the process are considered as a faulty state, the loss of data is arguable.

The described driver functions are integrated into the SK, cf. Figure 5.3. The global configuration pm_config() is called during the basic system setup. Afterwards the pm_cnt_config() can be used. It is invoked within the scheduling loop, during the process configuration. The resource capacity is taken out of the configuration for the respective process. As described above, int_hdlr_pm() is not actively called, i.e. its execution is triggered by the performance monitoring exception INT_PM, cf. Listing 5.1. If a process occupies multiple process frames within the major time frame the same mechanism applies, i.e. its resource capacity is initialised based on the offline-computed WCRA bound. Hence, the replenishment of the capacity of a process does not require any special handling.

While the described implementation is based on the bare-metal SK described in Section 5.2, a similar integration has been implemented in the context of the ARAMIS project by SYSGO for their PikeOS product and by Cassidian for VxWorks. PikeOS is a commercial Real-Time Operating System (RTOS), which has been used in multiple certified projects such as the Airbus A350 [SYSGO (2008a)] and Airbus A400M [SYSGO (2008b)]. SYSGO's implementation is based on a collaboration in the context of this thesis. Likewise, VxWorks is a commercial RTOS applied for certification projects, e.g. [Wind River (2004), GE (2009)].

5.3.3. Quality of Service Monitoring Extension

As described in Section 4.4, the QoS extension is based on a modified suspension routine. Instead of directly suspending a process, its capacity-extendible flag is checked. If the process is enabled for an extension, the re-calculation of the WCET bounds of the in-parallel scheduled processes is performed, based on the updated capacity values. The update is implemented via inter-core communication.

Accordingly, the int_hdlr_pm() callback function is extended. Further the additional function int_hdlr_dbell () is introduced, which implements the update of the resource capacity of a process. The inter-process communication is implemented based on the PowerPC doorbell mechanism. As described, a doorbell can be used to trigger an exception on other cores. Accordingly, the core which is calculating the capacity extension triggers the respective exception on all cores. In response the callback function shown in Listing 5.4 is executed on all cores. It reads the current value of the PMC and updates the configuration of the respective process. To signal the successful update, the global synchronisation variable update_finished is incremented. Since the increment is done in parallel by multiple cores, it has to be performed atomically.

```
1 // global synchronisation variable
2 unsigned int update_finished;
3
4 void int_hdlr_dbell(){
5     /* suspend current process */
6     core[this].current_process.pmc = 0x80000000 - reg_read(PMC_VAL);
7
8     /* signal update */
9     atomic_inc(update_finished);
10 }
```

Listing 5.4: *Doorbell exception handler.*

The inter-core communication is implemented such, that a minimum interval of $1ms$ is enforced between two subsequent communication requests. If that interval is not elapsed, the capacity

values from the last update are applied. As such, the communication mechanism is implemented much like a periodically triggered update as described in Section 4.4. However, if cores request updates less frequent than every $1ms$, the actual runtime overhead is much lower.

Listing 5.5 shows the extended suspension routine. Lines 73 to 77 cover the basic functionality as described previously, while the capacity extension is implemented in lines 25 to 69. The remaining lines 1 to 10 and 13 to 22 document the meaning of the system configuration parameters and the required global and local variables as well as their initialisation. Once the initialisation is finished, the capacity-extendible flag of the process is checked. If it enables the process for an extension the appropriate computation is performed. Therefor, an infinite loop is executed, which exits in either of the following cases:

E-1 a sufficient capacity extension has been calculated (lines 56 to 61),

E-2 the computed extension is not sufficient to re-iterate through the loop (lines 64, 65).

In Case **E-1** the process is allowed to be continued. On the other hand the variable suspend is still set to true for Case **E-2**, i.e. the process will be suspended. To compute the capacity extension c_ext, first the resource capacities are updated (lines 28 to 35). To avoid frequent updates with only minimal progress and to enforce the minimum interval between two communication requests, the time since the last update is compared against the predefined configuration value comm_threshold. If the time since the last update is not sufficiently large, the previous capacity values are applied. In effect, some computations of an extension can be based on older values. Since older values can only be higher than the actual ones, this is a safe design. However, since higher resource capacities cause increased isWCET bounds, this introduces overestimation which finally reduces the number of additional requests. To trigger the doorbell exceptions, the doorbell() call is used. Once all cores have updated their data, the time of the last update is set to the current time (line 34). Likewise, the overhead for executing the suspension routine is determined based on the statically known and the runtime parts (line 38). Afterwards, the maximum isWCET bound is computed as described in Section 5.4 (lines 41 to 45) and the number of active cores is determined (lines 48 to 50). Based on the complete data set, the capacity extension can be computed within line 53. Finally, the break conditions **E-1** and **E-2** are checked. Between two iterations the core waits for a given interval (line 67), in order to allow other cores to progress and increase the likelihood of a successful extension within the next iteration.

```
1  /* system configuration - conf
2   *
3   * .ncores              - number of physical cores
4   * .lat                 - access latencies d_i
5   * .comm_threshold      - time between two communications
6   * .susp_ovh_time       - temporal overhead of int_hdlr_pm()
7   * .susp_ovh_req        - WCRA bound of int_hdlr_pm()
8   * .ext_threshold       - min. number additional requests
9   * .ext_delay           - time to wait between iterations
10  */
11
12 void int_hdlr_pm(){
13     static unsigned int last_update;
14
15     bool suspend;
16     unsigned int t_pf, t_el, t_ovh, t_is, r, i, c_ext;
17
18
19     suspend = true;
20     t_pf = core[this].t_pf;
21     t_el = get_time() - core[this].current_process.start;
22     t_ovh = get_time();
```

```
23
24      /* check capacity-extendible flag */
25      if(core[this].current_process.extendible){
26          while(1){
27              /* trigger inter-process communication */
28              if(get_time() - last_update > conf.comm_threshold){
29                  update_finished = 0;
30                  doorbell();
31
32                  while(update_finished != conf.ncores);
33
34                  last_update = get_time();
35              }
36
37              /* determine execution time overhead */
38              t_ovh = (get_time() - t_ovh) + conf.susp_ovh_time;
39
40              /* compute max. isWCET bound */
41              t_is = 0;
42              for(i=0; i<conf.ncores; i++){
43                  r = is_wcet(i, core[i].current_process.wcet_s);
44                  t_is = (r > t_is) ? r : t_is;
45              }
46
47              /* determine number of active processes */
48              r = 0;
49              for(i=0; i<conf.ncores; i++)
50                  r = (core[i].current_process.capacity > 0) ? r + 1 : r;
51
52              /* compute capacity extension */
53              c_ext = (t_pf - t_el - t_ovh - t_is) / conf.lat[r]
54
55              /* reconfigure the PMC */
56              if(c_ext >= conf.ext_threshold + conf.susp_ovh_req){
57                  suspend = false;
58                  pm_cnt_cfg(c_ext);
59
60                  break;
61              }
62
63              /* check if extension allows another iteration */
64              if(c_ext < conf.susp_ovh_req)
65                  break;
66
67              wait(conf.ext_delay);
68          }
69      }
70
71      /* suspend process */
72      if(suspend){
73          /* suspend current process */
74          process_suspend(core[this].current_process);
75
76          /* switch ISR context to kernel */
77          next_process = kernel;
78      }
79  }
```

Listing 5.5: *Extended suspension routine, including the capacity extension.*

5.4. Interference-sensitive WCET Analysis

5.4.1. Core-local Analysis

The core-local analysis phase is required to determine bounds for the core-local WCET as well as the WCRA of a process. This section describes two different implementations. The first is based on static timing analysis, while the second relies on end-to-end measurements.

Static Analysis

The static analysis is based on a modified version of the commercially available timing analysis framework aiT, developed by AbsInt. The analysis has been extended in collaboration with AbsInt. The modifications comprehend the architecture model of the e500mc cores and modified micro-architecture and path analyses to compute the WCRA bound, as discussed in Section 4.3.1.

The architecture model is based on an early prototype of the e600 core. Architectural differences to the e500mc core have been derived from the processor manuals [FSL (2006), FSL (2011)]. The core-local memory hierarchy has been identified to possess the most severe differences. In particular, the e500mc contains L1 instruction and data caches, a unified L2 cache and mini caches for instruction and data, ILFB and DLFB, respectively. On the other hand, the architecture model provides a single cache level only. Thus, the model can either abstract the L1 caches or the mini caches by varying the cache size and the associativity. However, analyses with both configurations indicate a significant impact of both cache levels, i.e. although ILFB and DLFB contain only two and five cache lines, respectively, their impact on the analysis results cannot be neglected. As a result, the computed bounds will contain a significant overestimation. Nevertheless, it shall be understood, that the development of an accurate architecture model is very time consuming and thus not feasible for a research project. Having this in mind, the prototype model is considered sufficient for the evaluation of the approach.

The micro-architecture analysis has been extended to compute a bound for the number of off-core memory requests per basic block. Therefor, the classification of memory requests as cache hit, miss or unknown, which is available through the cache analysis, is used to distinguish core-local (cache hit) and off-core (cache miss and unknown) memory requests. In order to provide a safe approximation, accesses which are classified as unknown are treated as off-core requests. Further, the analysis does not differentiate read and write requests, i.e. a basic block gets assigned a single WCRA bound, containing the numbers for both request types. As a consequence, both kinds of requests are considered with the same access latencies, although they might be different in reality.

Finally, the path analysis has been extended to compute the overall WCRA bound of an application. Therefor, an additional ILP has been applied, which optimises towards the path with the highest number of off-core memory requests. As discussed in Section 4.3.1, the computation of the path with the longest execution time already accounts for off-core memory access latencies. It has been outlined, that those latencies should be stripped from the resulting core-local timing bound to increasing the accuracy of the timing analysis,. So far, the analysis does not implement such a mechanism. Hence, the resulting isWCET bound includes additional overestimation since an unknown number of requests is accounted twice.

End-to-end Measurement

As an alternative to static analysis, end-to-end measurements have been implemented. This includes timing and resource measurements for entire application executions. The timing measure-

ments are based on the core timebase register. Since preemptions due to external events and other processes can be excluded in the underlying SK, the timing measurements yield precise results. Similarly, off-core requests are measured for entire executions. They are measured based on the core PMCs as described in Section 5.3. As for the static analysis, read and write transactions cannot be distinguished. In contrast to the static analysis, each timing measurement also traces the resource usage. Hence, the latencies for off-core requests can be stripped from the measured core-local execution times. The core-local measurements are taken in isolation, i.e. all but one core are inactive. Accordingly, the minimal measured access latency is used to strip the request latencies.

5.4.2. Interference-delay Analysis

The computation of the interference delay is covered in a separate analysis step. As discussed above, the off-core memory is considered as a single-channel resource. Hence Equation 4.8 applies for the calculation of the interference-delay. Further, since a single resource r_{NoC} is considered, Equations 4.36 to compute the resulting isWCET bound can be simplified as shown in Equation 5.1.

$$t_{is}(p_h) = t_{s,p_h} + d_{|P_{||}|} \cdot c_{p_0,r_{NoC}} + \sum_{i=1}^{h} \left[d_{|P_{||}|-i} \cdot \left(c_{p_i,r_{NoC}} - c_{p_{i-1},r_{NoC}} \right) \right] \tag{5.1}$$

The interference-delay analysis step is covered in the function `is_wcet()`, as shown in Listing 5.6. The arguments to the function are the index of the process whose isWCET bound shall be computed and the respective core-local WCET bound. The resource capacities of the processes are deduced from the `core[i].current_process.pmc` variables. In the first step they need to be sorted in ascending order and stored into the `capa` array (line 6). Afterwards Equation 5.1 is applied, whereat the access latencies d_i are statically defined in the system configuration (lines 9 to 13). Finally, the isWCET bound is returned (line 16).

```
1  unsigned int is_wcet(unsigned int idx, wcet_s){
2      unsigned int capa[conf.ncores], id, i;
3
4
5      /* sort process capacities */
6      sort(core, capa);
7
8      /* compute interference-delay (id)*/
9      id = conf.lat[conf.ncores - 1] * capa[0];
10
11     for(i=1; i<=idx; i++)
12         id += conf.lat[conf.ncores - 1 - i] * \
13               (capa[i] - capa[i - 1]);
14
15     /* return resulting isWCET bound */
16     return wcet_s + id
17 }
```

Listing 5.6: *isWCET bound computation.*

6. Evaluation

In this chapter the presented approaches are evaluated. This covers the mechanics of the runtime monitoring, the core-local and isWCET analysis and the QoS extension. The evaluation is based on the implementation described in Chapter 5. That is, the Freescale P4080 being the target platform, running a bare-metal operating system layer in combination with static timing analysis based on AbsInt's aiT framework and measurement-based analysis using end-to-end measurements. Further, the EEMBC Autobench benchmark suite [EEMBC (2013)] is used to construct different application scenarios. As the foundation for the analysis, first the relevant target architecture parameter, i.e. the off-core memory access latencies, are derived. Additionally, the selected benchmarks of the Autobench suite are characterised.

6.1. Architecture Analysis

The architecture analysis has been identified as part of the hardware development, supplying input information required for timing analysis, cf. Section 4.6. Parameters, such as cache hierarchy and replacement policies have been summarised in Section 5.1. Further parameters like clock rates are derived from the platform and board manuals. The evaluation has been performed on the revision 1.0 of the P4080. The e500mc cores are clocked at 1.2 GHz, while the CoreNet and main memory controllers are operated with 600 MHz. As an additional step, the off-core memory latencies are determined to perform the isWCET bound computation. They are derived by measurements, considering the interference by PEs and DMA-I/O devices. In the following section the basic measurement approach is described. Afterwards, the results for concurrent off-core requests by PEs and DMA-I/O devices are extracted.

6.1.1. Measurement Setup

To measure the impact of concurrent off-core memory accesses a synthetic interference application (*synthetic Interference (sInt)*) has been created which is targeted towards maximum load on the NoC. For that purpose the application constantly issues load/store instructions, which are aligned such, that the usage of core-local caches is minimized. Since the application is running bare-metal, the synchronisation as well as the activities of all PEs can be controlled in a way, such that no preemptions or intermediately scheduled applications disturb the measurements. The schematics of sInt are shown in Listings 6.1 and 6.2. Listing 6.1 shows the application body, while Listing 6.2 depicts the details of the access pattern.

The body of the applications contains the main loop which is executed within one thread per core. At the beginning of the loop the local caches are flushed and invalidated, such that no valid entries are retained. Afterwards a barrier operation is used to synchronise all cores. The barrier is implemented based on the doorbell and wait mechanism of the e500mc cores. In particular, all, but the last core execute the `wait` instruction, which puts the core in a idle mode, pausing execution until an exception occurs. The last core issues a doorbell, which wakes all cores at the same point in time. Thereby, a very tight synchronisation of up to the same instruction is

achieved. This has been verified by an external debugger, that halts the system once one of the cores reaches the first instruction after the wait instruction. Once the synchronisation has been performed, the actual measurement code is executed. This loop executes for NMEAS iterations to mitigate timing differences and gain statistically relevant results. Such timing difference can for instances be caused by clock domain synchronisations and different DDR states. NMEAS is set to 100, since no significant changes in the dataset could be observed for larger values.

```
1 for(i=0; i<NMEAS; i++){
2       flush_invalidate_caches();
3       barrier();
4
5       time();
6       meas_loop(OPERATION, NBYTES, GAP, ADDR);
7       time();
8 }
```

Listing 6.1: *Main body of the sInt application.*

The parameters to the measurement are OPERATION, ADDR, GAP and NBYTES.

OPERATION: The application can either issue load or store operations, that is, OPERATION is set to read or write. During execution the PEs are virtually divided into master and slave cores, whereat only the master reports the measured execution time, while the slaves cause interference to the master's off-core memory requests. The OPERATION parameter for master and slave cores can be chosen independently, while the remaining parameters are equal for all cores. Thus, the application can be executed in the following four master-slave configurations: read-read, read-write, write-read and write-write.

ADDR: The ADDR parameter defines the target address of the load/store instructions. Through the physical memory layout this also defines the target memory device. Initially, the measurements were targeted towards the platform caches, which can be configured as Static Random Access Memory (SRAM) and accessed as dedicated address space. To acquire off-core latencies the main memory is configured as target memory. The addresses of different cores are configured such, that each core accesses a private memory segment. Hence, additional overhead due to coherency is avoided.

GAP: The GAP parameter is essential to control the usage of the core-local caches. Since the measurements are intended to stress the off-core memory, core-local requests shall be avoided. Local-caches can only be disabled to a certain extend, i.e. the mini caches DLFB and ILFB cannot be disabled. Hence, it has to be ensured that subsequent requests do not hit cache lines which are already present in the DLFB. Considering a cache line size of $64B$, the byte offset between two consecutive accesses is set $64B$. The byte offset is further referred to as GAP parameter. Figure 6.1 shows different GAP values in relation to a $64B$ cache line.

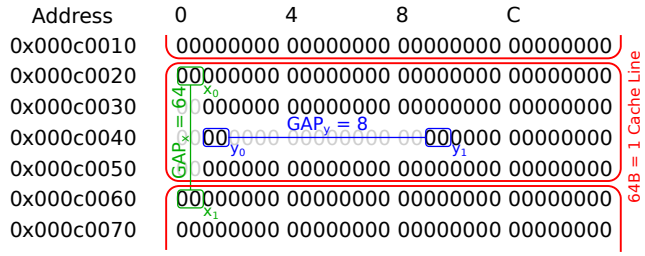


Figure 6.1.: *Access Pattern with different GAP values ($8B$ and $64B$) in relation to a cache line of size $64B$.*

NBYTES: The amount of load/store operations for one call of the meas_loop is defined with the NBYTES parameter. The used configuration of $4096B$ per iteration is defined based on the number of PEs (8), the size of the platform cache ($2MB$) and the value of the GAP parameter ($64B$). The platform cache size is considered, since the measurements are initially designed towards the platform cache and a comparison to the main memory.

The actual measurement loop, Listing 6.2, contains a loop of load/store instructions (lines 6 to 17), surrounded by initialisation (line 2) and cleanup code (line 23) for stack handling.

```
1 #define meas_loop
2     __asm_meas_init
3
4     mr      ADDR, %0;
5
6     1:
7     stb     0x2a, 0x000(ADDR);
8     stb     0x2a, 0x040(ADDR);
9     stb     0x2a, 0x080(ADDR);
10    stb     0x2a, 0x0c0(ADDR);
11    stb     0x2a, 0x100(ADDR);
12    stb     0x2a, 0x140(ADDR);
13    stb     0x2a, 0x180(ADDR);
14    stb     0x2a, 0x1c0(ADDR);
15    add.    ADDR, ADDR, __gap_x;      // __gap_x = GAP * 8
16    cmpw    7, __addr_last, ADDR;     // __addr_last = ADDR + NBYTES * GAP
17    bgt     7, 1b;
18
19    __asm_meas_clean
```

Listing 6.2: *sInt application inner loop for store operations and a GAP size of* $64B$.

During measurements the number of active PEs is increased from one to eight. If a PE does not contribute to a measurement it waits within a barrier, i.e. waiting in a halting state. Hence, the inactive PEs will not interfere with the active ones.

6.1.2. Processing Element Latency Dependence

First, the maximum impact of multiple PEs, that concurrently access off-core memory is quantified. Therefor, the described measurement approach is applied. Local-caches and branch prediction are disabled to avoid any known positive pipeline effects. The results are summarised in Table V and Figure 6.2. Figure 6.2 shows the access latency for each combination of read and write accesses (y axis) over the number of active cores (x-axis). As described in Chapter 5, read and write accesses are not distinguished in the current implementation. To safely account the access latencies the measured maximum values are used. They are highlighted in Table V using a bold font.

Table V.: *Maximum off-core memory access latencies depending on the number of concurrent PEs, with overall maximum values highlighted bold.*

	Latency [*cycles*]							
Cores	1	2	3	4	5	6	7	8
read-read	**41**	70	106	140	226	342	364	483
read-write	41	75	171	269	296	439	460	604
write-read	39	66	98	136	225	284	319	422
write-write	39	**164**	**245**	**463**	**517**	**737**	**784**	**1007**

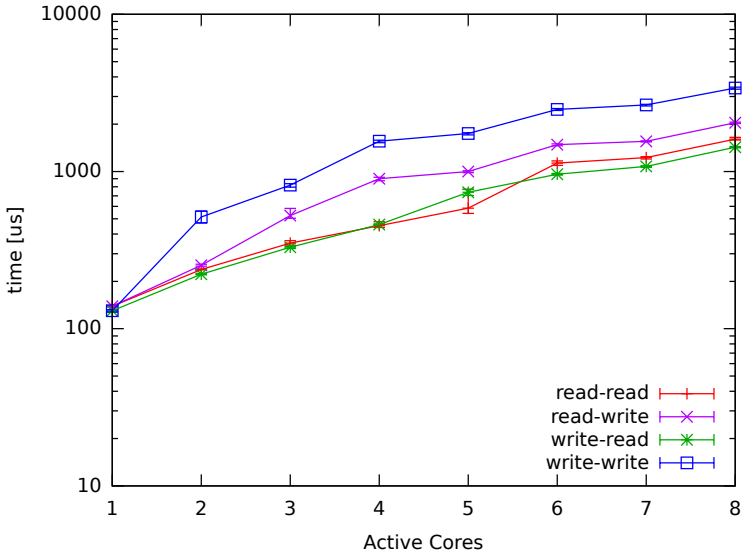


Figure 6.2.: *Maximum off-core memory access latencies over the number of active cores.*

Proving the measured latencies against Equation 4.2 reveals, that the relation is not fulfilled for all pairs d_i, d_{i+1}. That means, relative access latencies do not constantly increase. On the other hand, the results do not conform to Equation 4.17 for multi-channel resources. Hence, solely based on the results, the off-core memory cannot be classified as single- or multi-channel resource. However, considering the SoC architecture, the usage of a single memory controller and the rather small deviations between the results and Equation 4.1, the off-core memory is treated as a single-channel resource. The deviations between the measured latencies and Equation 4.1 can most likely be explained with the complexity of the path from the core pipeline to the main memory. That is, without detailed information on the implementation of the system it is very hard or even impossible to trigger worst-case behaviour in every single part of that hierarchy. For the same reasoning the acquired values shall not be considered as absolute worst-case values. While they are sufficient for the evaluation of the approach, the architecture analysis for a real system needs to be more exhaustive and should rather be based on detailed platform design information or an accurate model of the memory hierarchy, instead of pure measurements.

In a similar manner as for the maximum latencies, the minimum latencies are acquired. Instead of stepping through the whole address space of NBYTES · GAP bytes, the meas_loop cycles over an address space of 6· GAP. With respect to the main memory controller, this should cause viewer row switches, reducing the overall latency to a minimum. The results are shown in Table VI, the overall minimum values are highlighted with a bold font.

Table VI.: *Minimum off-core memory access latencies depending on the number of concurrent PEs, with overall minimum values are highlighted bold.*

	Latency [*cycles*]							
Cores	1	2	3	4	5	6	7	8
read-read	32	48	64	84	104	124	144	172
read-write	32	48	84	120	160	204	252	288
write-read	**20**	**24**	**36**	**40**	**44**	**48**	**60**	**76**
write-write	20	48	68	88	76	120	184	220

6.1.3. DMAI/O Device Latency Dependence

As described in Section 4.1, beneath PEs, DMA-I/O devices compete for the shared NoC. While DMA-I/O devices can be very different, their effect on the NoC and thus on the rest of the system can be abstracted as additional requests due to their DMA capabilities. Accordingly, to quantify the effect on the target platform, the built-in DMA controllers are used. To measure their impact, the sInt application is extended to additionally configure DMA transfers, which are scheduled in parallel to the requests of the PEs. Therefore, the last core does not execute the described meas_loop, but instead controllers the DMA controllers. The P4080 provides two DMA controllers with four channels each. At a time only one of the channels per controller can access the NoC. Each channel is configured to transfer a data block of $4MB$ in size. As for the measurements of the PE interference, the number of active DMA channels is constantly increased until all of them are active. The results are shown in Figure 6.3 and Table VII. The particular impact of the DMA transfers can be seen for the active masters values from seven to fourteen. Since the results for concurrent PEs are equal to the results in Table V, they are shortened in Table VII.

Table VII.: *Impact of DMA-I/O devices on the off-core memory access latency.*

	Latency [*cycles*]										
Cores	1	...	7	8	9	10	11	12	13	14	15
read-read	41		364	500	501	502	500	501	502	495	497
read-write	41		460	560	556	556	559	620	618	618	617
write-read	39		319	521	519	520	523	528	523	531	525
write-write	39		784	902	904	904	902	988	994	997	993
	cf. Table V			DMA Interference							

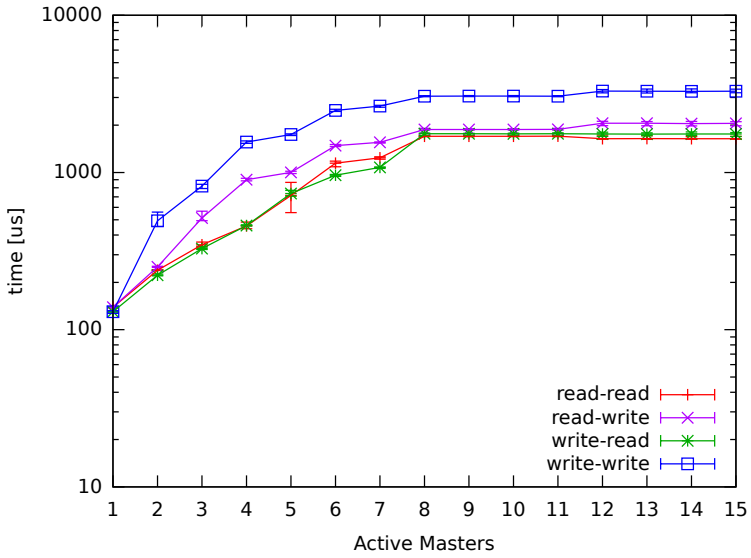


Figure 6.3.: *Impact of DMA-I/O devices on the off-core memory access latency.*

As suspected, the results prove a similar impact of DMA requests than concurrent requests by PEs. When assessing the results one shall bear in mind, that even though eight DMA channels are plotted, only two of them can access the shared memory in parallel.

6.2. Benchmark Characterisation

The evaluation of the isWCET analysis and the QoS extension are based on a subset of the EEMBC Autobench benchmark suite. Although those benchmarks are not designed for timing analysability, they are chosen as a representative set of real-world applications. Each of the selected benchmarks is ported to the target platform and SK. Since the benchmark suite is not designed towards multi-core systems the main benchmark functions and the initialisation routines needed to be adapted in order to be executable on multiple cores. The available code for performance measurements within the benchmarks has been replaced to fit the needs of the evaluation.

Table VIII shows the configuration and characterisation of the selected benchmarks. It lists the benchmark name, the number of iterations within its main function, the multi-core sensitivity ($msens$) and the number of off-core requests per micro second. The multi-core sensitivity describes the execution time increase, when comparing isolated and concurrent execution. In isolation the respective benchmark is executed on one core, while the remaining cores execute a wait instruction within the scheduler loop. For concurrent execution different workloads are scheduled in parallel to the measured benchmark. To cover the benchmark behaviour under different interference configurations three different workloads are executed concurrently. The listed results reflect the respective maximum over all measurements. Firstly, each benchmark is executed in parallel to instances of them self. Secondly, the sInt application is scheduled on the remaining seven cores, while for the third case the cacheb benchmark is used. Over the available EEMBC benchmarks cacheb is the one that is designed for the highest amount of cache misses and thus off-core requests. The number of off-core memory requests has been measured using the core PMCs described in Section 5.3. The measurements have been taken once with enabled L1 and once with enabled L1 and L2 caches. Based on the multi-core sensitivity and the accesses over time the locality of the benchmarks is classified. A high locality describes a benchmark which executes most of the time on core-local resources and thus is relatively insensitive against concurrent execution. Accordingly, benchmarks with a low locality are sensitive to concurrent execution and issue a high amount of off-core requests over time. Finally, based on the benchmark algorithms [EEMBC (2013)], an exemplary target application is assigned. For instance, Fast Fourier Transform (FFT) algorithms are intensively used for object detection in images [Dubout and Fleuret (2012)].

Table VIII.: *Benchmark characterisation, showing the number of iterations, the multi-core sensitivity msens and the resource requests over time a/t for different cache configurations as well as the locality and exemplary application domains.*

Benchmark	Iterations	L1 Caches		L1, L2 Caches		Locality	Application Domain
		$msens$	a/t $[1/us]$	$msens$	a/t $[1/us]$		
cacheb	100000	15.53	86.4	8.57	69.0	Low	Data Loading
iirflt	10000	10.09	43.2	1.14	1.7	Medium	Audio Processing
rspeed	100000	14.25	105.8	16.16	99.1	Low	Control
a2time	10000	12.95	83.0	14.90	82.5	Low	Control
bitmnp	1000	1.50	4.2	1.06	0.9	High	Display
tblook	10000	2.03	8.2	1.43	3.7	High	Image Processing
matrix	10	2.87	10.5	1.08	0.8	High	Optimisation
aifftr	4	8.82	30.4	1.05	0.1	Medium	Image Processing

6.3. Resource Usage Enforcement

In this section the functional behaviour of the resource usage enforcement shall be demonstrated to ensure, that the partitioning mechanism works as desired. For that purpose three different scenarios are constructed:

Sc-1 isolated: where core pe_0 executes the reference benchmark, while the remaining cores are inactive,

Sc-2 interfered: similar to **Sc-1**, but with interference by additional benchmarks executed on cores pe_1 to pe_7. To increase the interference, the concurrent benchmarks are executed in a loop which exits once the reference benchmark on pe_0 terminates.

Sc-3 partitioned: similar to **Sc-2**, but with enabled resource usage enforcement on all cores.

As reference benchmark bitmnp has been chosen, while the remaining benchmarks are scheduled on cores pe_1 to pe_7 to introduce interference for Scenarios **Sc-2** and **Sc-3**. To intense the effect of interference, all caches are disabled, i.e. only ILFB and DLFB are enabled. It shall be understood that this is solely for demonstration purposes. For Scenario **Sc-3**, the benchmark resource capacities c_{cfg}, cf. Table IX, are acquired by measurements. The results for each scenario are shown in Figures 6.4, 6.5 and 6.6, respectively. Each figure shows a single plot per active core, whereat every plot depicts the number of off-core memory requests c over time. Since for Scenario **Sc-1** only the first core is active, the others are not shown. Red triangles indicate the suspension of a benchmark due to a resource capacity violation. Additionally, the observed execution times t_{obs} and off-core memory requests c_{obs} are summarised in Table IX.

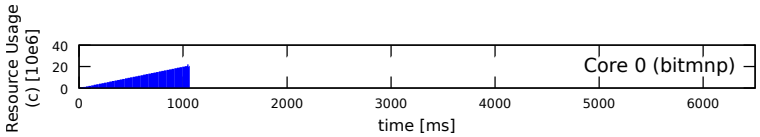


Figure 6.4.: *Observed resource usage c over time for Scenario **Sc-1**.*

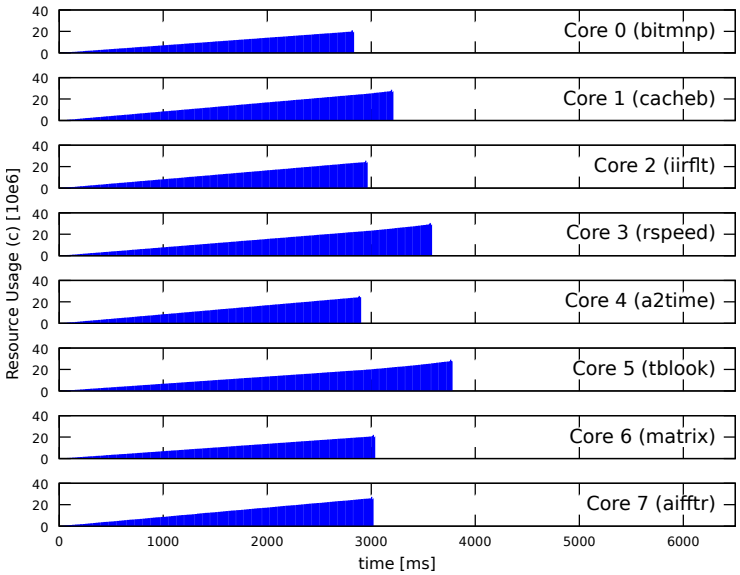


Figure 6.5.: *Observed resource usage c over time for Scenario **Sc-2**.*

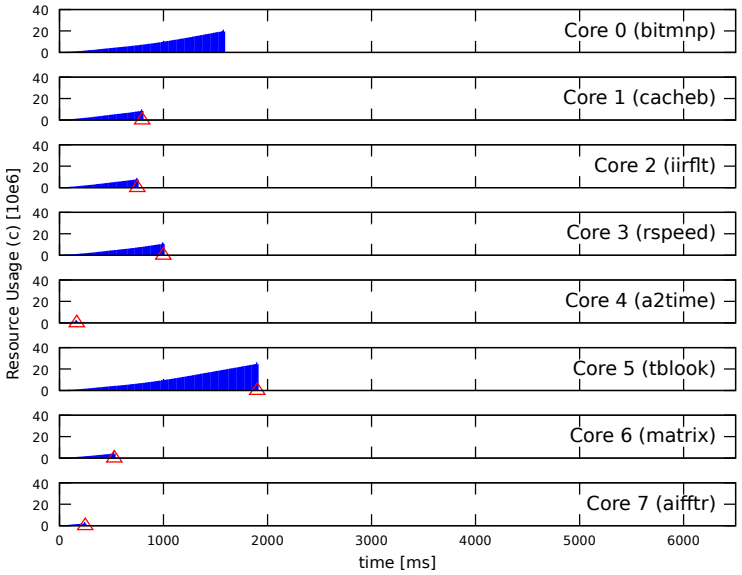


Figure 6.6.: *Observed resource usage c over time for Scenario **Sc-3**.*

Table IX.: *Observed execution times t_{obs}, off-core memory requests c_{obs} and configured capacities c_{cfg} for Scenarios **Sc-1**, **Sc-2** and **Sc-3**.*

Benchmark	Scenario **Sc-1**		Scenario **Sc-2**		Scenario **Sc-3**			
	t_{obs} $[ms]$	c_{obs}	t_{obs} $[ms]$	c_{obs}	t_{obs} $[ms]$	c_{obs}	c_{cfg}	$c_{obs} - c_{cfg}$
bitmnp	1053	20906919	2823	19948232	1581	20048632	21886890	-
cacheb	-	-	3201	27498647	795	8687353	8687344	9
iirflt	-	-	2953	24406444	749	7907687	7907677	10
rspeed	-	-	3537	29383240	998	10845092	10845083	9
a2time	-	-	2890	24346857	166	1377826	1377816	10
tblook	-	-	3770	27914081	1902	25021973	25021964	9
matrix	-	-	3026	20869693	527	4378300	4378291	9
aifftr	-	-	3010	26112340	248	2187481	2187468	13

The comparison of the results for one shows the impact of interference and also the correct functioning of the resource usage enforcement. While the bitmnp benchmark in isolation finishes after $1053ms$, its execution time is increased to $2823ms$ for Scenario **Sc-2**. For Scenario **Sc-3** the benchmarks on cores pe_1 to pe_7 are suspended once they reach their resource capacity. Accordingly, the execution time of the reference benchmark is reduced to $1581ms$. Table IX further shows the difference between the configured resource capacity c_{cfg} and the observed value c_{obs} for Scenario **Sc-3**. It can be seen, that the observed capacities slightly deviate from the configured values by about 9 to 13 requests. This deviation is a combination of the overhead of suspending a process and the reaction time of the overflow exception. The overhead for executing the suspension routine is quantified in Table X, it shows statically analysed as well as measured results for different cache configurations. Since the architecture model used for the static analysis can only model a single cache layer, it only reports results for disabled L1 and L2 caches, i.e. solely ILFB and DLFB enabled. As can be seen, the overhead listed in Table X is much higher than the deviation shown in Table IX. This is explained by the implementation of the ISR, which freezes the PMCs in the beginning, i.e. most of the requests during the execution of the ISR

are not monitored. This is essential to avoid another overflow exception from being triggered immediately.

Table X.: *Overhead for the suspension routine, showing the execution time t and the off-core memory requests c for different cache configurations and analysis methods.*

	Static Analysis		Observed	
	t $[us]$	c	t $[us]$	c
DLFB, ILFB	30	714	25	203
L1 Caches	-	-	14	129
L1, L2 Caches	-	-	10	67

Additionally to the overhead for executing the ISR, a certain amount of time is spent between the occurrence of the overflow event and the actual execution of the ISR. This time is generally referred to as interrupt latency. The e500mc manual [FSL (2011)] lists a latency of $\leq 10ms$, unless a guarded load or a cache-inhibited stwcx. instruction is in the last completion queue entry of the core pipeline. For the latter cases the latency is determined by the target memory location of the instruction.

6.4. Core-local Analysis

The core-local analysis is evaluated towards two aspects. Firstly, the impact of the limited capabilities of the architecture model, applied for the static analysis, and secondly, the overestimation of the static analysis compared to measured execution times are quantified.

6.4.1. Architecture Model Limitations

As described in Section 5.4.1, the architecture model used for the static analysis is a prototypical version of the e500mc core, which is based on an earlier e600 model. As described in Chapter 5, this prototype provides a single cache level only, i.e. in comparison to the e500mc hardware, which contains three levels (ILFB, DLFB, L1 and L2), two cache levels are missing. This part of the evaluation is used to quantify the impact of the missing cache levels on the core-local analysis results. Table XI shows the computed core-local WCET bounds (t_{s,p_i}) and WCRA bounds (c_{p_i}) for the following configurations of the static analysis framework:

SA-1 All load/store instructions are handled as cache misses, i.e. the configuration models a core without any caches.

SA-2 The available cache level is used to represent the ILFB and DLFB, i.e. two fully-associative, $64B$ cache lines for the instruction and five fully-associative $64B$ cache lines for the data side.

SA-3 The available cache level is used to model the L1 instruction and data caches, i.e. 64 sets, 8-way associative, $64B$ cache lines for the instruction and the data side, respectively.

Based on the individual results, Table XI also lists the reduction for Configurations **SA-2** and **SA-3** relative to Configuration **SA-1**.

The comparison of the achieved reductions with ILFB, DLFB and L1 caches clearly shows, that the effect of the L1 caches is higher, which is expected due to the significantly larger cache size. However, the impact of ILFB and DLFB cannot be neglected as well. As a conclusion of this comparison it shall be remembered, that the missing cache levels in the architecture model are

Table XI.: *Static analysis results for different cache analysis configurations of the architecture model, showing the core-local WCET bounds t_{s,p_i} and the resource capacities c_{p_i}.*

Benchmark	**SA-1**		**SA-2**		**SA-3**		Reduction [%] $100 - \frac{\textbf{SA-2} \cdot 100}{\textbf{SA-1}}$		Reduction [%] $100 - \frac{\textbf{SA-3} \cdot 100}{\textbf{SA-1}}$	
	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{s,p_i}	c_{p_i}	t_{s,p_i}	c_{p_i}
cacheb	832	20.5	424	9.9	318	7.2	49.0	51.7	61.8	64.7
iirflt	1238	30.3	532	13.3	399	9.6	57.0	55.9	67.8	68.3
rspeed	1624	38.9	905	19.5	398	9.9	44.3	49.9	75.5	74.6
a2time	303	7.3	162	3.3	47	1.2	46.5	54.9	84.6	83.2
bitmnp	7468	172.8	2586	54.5	2052	40.7	65.4	68.5	72.5	76.4
tblook	5740	138.9	2664	63.2	1249	29.5	53.6	54.5	78.2	78.7
matrix	12707	301.8	6451	137.4	4145	78.0	49.2	54.5	67.4	74.1
aifftr	19519	478.6	7882	197.8	4838	122.5	59.6	58.7	75.2	74.4

a cause for major overestimation when comparing the static analysis results with the measured behaviour of a system with fully enabled caches.

6.4.2. Overestimation

To assess the quality of the static analysis results, their overestimation is quantified. As discussed in Section 2.4, the real WCET of an application is generally unknown, thus, it is not possible to exactly quantify the overestimation of analysis results. However, it is a common approach in literature to compare the statically computed bounds with the maximum observed values, cf. [Souyris et al. (2005), Thesing et al. (2003)]. Accordingly, Table XII shows the comparison of the static bounds for execution time t_{s,p_i} and resource capacity c_{p_i}, with the observed maximum execution time t_{obs} and number of off-core memory requests c_{obs}. Considering the limitation of the architecture model with respect to the abstracted cache levels, the measured values are obtained with disabled caches. In consequence the measured values are comparable to the static analysis with Configuration **SA-2**. Furthermore, since in the current version of the static analysis the off-core requests on the worst-case timing path cannot be stripped from the timing bounds, the measured timing bounds also include the access latencies. To quantify the overestimation Table XII lists the deviation of the statically analysed bounds relative to the observed values.

As can be seen from the results, the absolute overestimation greatly depends on the benchmark, which is as expected. Over all benchmarks the overestimation ranges from about 14% to 8900%, while the overestimation for execution time and off-core requests for a benchmark is roughly within the same order of magnitude. The dramatic overestimations for the matrix and aifftr benchmarks are mostly explained through their code structure, which contains hard to analyse triangular loops. Triangular loops, are nested loop structures, where the number of iterations are related to each other. This often causes either huge state space or reduced tightness, i.e. overestimation, depending on the applied trade-off. Since the size of the state space and the amount of analysis time was not maintainable, the trade-off for the evaluations has been chosen towards reduced analysis complexity, which ultimately causes significant overestimations for the matrix and aifftr benchmarks.

Table XII.: *Overestimation of core-local static analysis, comparing the bounds for WCET t_{s,p_i} and off-core requests c_{p_i} with the observed maximum execution time t_{obs} and off-core requests c_{obs}.*

Benchmark	Static Analysis		Observed		Deviation $\frac{static \cdot 100}{observed} - 100$	
	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{obs} $[ms]$	c_{obs} $[10^6]$	t [%]	c [%]
cacheb	424	9.9	372	8.7	14.0	13.9
iirflt	532	13.3	363	9.3	46.7	43.0
rspeed	905	19.5	450	14.9	101.2	30.7
a2time	162	3.3	68	1.9	138.2	68.6
bitmnp	2586	54.5	1082	22.7	139.1	140.5
tblook	2664	63.2	1341	29.6	98.7	113.7
matrix	6451	137.4	231	4.4	2695.8	3054.1
aifftr	7882	197.8	89	2.2	8714.5	8874.4

6.5. Interference-sensitive Analysis

The goal of this part of the evaluation is to assess the benefit of using the isWCET analysis over straight forward approaches. As reference the minimum-Contention (minCont) and maximum-Contention (maxCont) approaches are considered. For maximum contention all off-core requests are assumed to suffer full interference by the remaining cores. This approach is especially required if no assumptions on the in-parallel scheduled applications and the underlying software can be made. It is for instance applied in [Schliecker et al. (2010), Yun et al. (2012), Behnam et al. (2012)]. The execution time t_{max} for maxCont is computed based on Equation 6.1. The latency $d_{\left|P_{||}\right|}$ depends on the number of in-parallel scheduled processes $\left|P_{||}\right|$, i.e. $\left|P_{||}\right| = 8$ for the P4080.

$$t_{max}\left(p_h\right) = t_{s,p_h} + d_{\left|P_{||}\right|} \cdot c_{p_h,r_{NoC}} \tag{6.1}$$

The minCont approach represents the single-core execution of each process, hence the off-core memory access latency for a single active PE represented by d_1, is considered, cf. Equation 6.2.

$$t_{min}\left(p_h\right) = t_{s,p_h} + d_1 \cdot c_{p_h,r_{NoC}} \tag{6.2}$$

The isWCET bounds are computed with Equation 5.1. The applied access latencies are highlighted with a bold font in Table V.

In the following the results for three different setups are presented.

IS-1 Static core-local analysed, whereat the available cache level models ILFB and DLFB (Table XIII). This setup is used to evaluate the results of the static analysis.

IS-2 Measurement-based core-local analysis with disabled caches (Table XIV). Since caches are disabled, this setup is the reference for the comparison to the statically analysed setup.

IS-3 Measurement-based core-local analysis with enabled L1 and L2 caches (Table XV). In contrast to Setup **IS-2**, this represents a more realistic configuration.

It shall be noted, that the latencies for the off-core requests have been stripped from the timing bounds for Setup **IS-3**. That is, the off-core memory requests which have been traced during the maximum observed execution timing, have been accounted with the minimum latency, cf. Table VI, and subtracted from the core-local execution time. Hence, the acquired data represent the core-local analysis as described in Section 4.3.1. For Setup **IS-2** the latencies have not been stripped, to retain the same basis for comparison to Setup **IS-1**.

Each of the Tables XIII to XV shows the core-local bounds for execution time t_{s,p_i} and the number of off-core requests c_{p_i}, which are used to compute the minCont, maxCont and isWCET bounds t_{min}, t_{max}, t_{is}, respectively. To compute the isWCET bounds, it is assumed that all of the listed benchmarks are scheduled in parallel. Based on the computed bounds, the deviation between t_{is} and t_{min}, as well as the achieved reduction for t_{is} in comparison to t_{max} are shown.

Table XIII.: *isWCET assessment for Setup **IS-1**, showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}, comparing the isWCET bound t_{is}, the maxCont t_{max} and the minCont t_{min}.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{min} $[ms]$	t_{max} $[ms]$	t_{is} $[ms]$	Deviation [%] $\frac{t_{is} \cdot 100}{t_{min}} - 100$	Reduction [%] $100 - \frac{t_{is} \cdot 100}{t_{max}}$
cacheb	424	9.9	762	8732	7500	883.9	14.1
iirflt	532	13.3	988	11728	9722	884.1	17.1
rspeed	905	19.5	1572	17276	12751	711.4	26.2
a2time	162	3.3	274	2907	2907	961.9	0.0
bitmnp	2586	54.5	4448	48329	27937	528.0	42.2
tblook	2664	63.2	4824	55711	29792	517.6	46.5
matrix	6451	137.4	11144	121725	43714	292.2	64.1
aifftr	7882	197.8	14641	173883	47210	222.5	72.8

Table XIV.: *isWCET assessment for Setup **IS-2**, showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}, comparing the isWCET bound t_{is}, the maxCont t_{max} and the minCont t_{min}.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{min} $[ms]$	t_{max} $[ms]$	t_{is} $[ms]$	Deviation [%] $\frac{t_{is} \cdot 100}{t_{min}} - 100$	Reduction [%] $100 - \frac{t_{is} \cdot 100}{t_{max}}$
cacheb	372	8.7	669	7668	5363	701.8	30.1
iirflt	363	9.3	681	8190	5598	721.6	31.6
rspeed	450	14.9	960	12976	6829	611.5	47.4
a2time	68	1.9	134	1696	1696	1163.0	0.0
bitmnp	1082	22.7	1856	20099	8518	358.9	57.6
tblook	1341	29.6	2351	26165	9013	283.3	65.6
matrix	231	4.4	380	3886	3352	783.3	13.7
aifftr	89	2.2	165	1939	1890	1047.3	2.5

In contrast to the results in Table XIV, the results for enabled caches represent an optimised system. Since the benchmarks are relatively small, larger portions of code and data fit into the local caches, which in turn reduces the number of off-core memory requests. As a consequence the resulting timing bounds are significantly smaller.

The results for Setups **IS-2** and **IS-3** are representative for different kinds of systems. Whereas the applications in Setup **IS-2** do not entirely fit into the core-local caches or do not properly utilise them, the applications in Setup **IS-3** have a higher cache utilisation. As a consequence, Setup **IS-2** produces a higher amount of off-core requests, which in turn causes higher timing bounds. From a use case perspective the comparison of those results indicate the impact of the system configuration and the application characteristics on the resulting timing bounds.

Considering the results of all three setups, it can be seen, that the isWCET approach can greatly reduce the timing bounds in comparison to the maxCont approach. Given by the computation of

Table XV.: *isWCET assessment for Setup **IS-3**, showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}, comparing the isWCET bound t_{is}, the maxCont t_{max} and the minCont t_{min}.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	t_{min} $[ms]$	t_{max} $[ms]$	t_{is} $[ms]$	Deviation [%] $\frac{t_{is} \cdot 100}{t_{min}} - 100$	Reduction [%] $100 - \frac{t_{is} \cdot 100}{t_{max}}$
cacheb	34	2.3	114	1996	493	333.2	75.3
iirflt	57	0.1	60	136	116	92.0	15.0
rspeed	53	5.3	233	4468	612	162.8	86.3
a2time	7	0.6	29	524	231	710.6	55.9
bitmnp	149	0.1	154	262	225	46.7	14.1
tblook	108	0.4	122	449	289	136.4	35.6
matrix	21	0.0	21	35	32	50.0	8.5
aifftr	11	0.0	11	11	11	7.1	0.0

the isWCET bound, the benchmark with the fewest number of off-core requests suffers maximum contention for all of them, hence a reduction for that benchmark is not possible, cf. a2time in Tables XIII and XIV, and aifftr in Table XV. The reduction for the other benchmarks depends on the difference in the number of off-core requests. The higher the differences, the higher the potential reduction. To ease the comparison of the reduction, Figure 6.7 plots the isWCET bounds in relation to those for the maxCont. As can be seen, the isWCET bound computation produces tighter bounds for each setup.

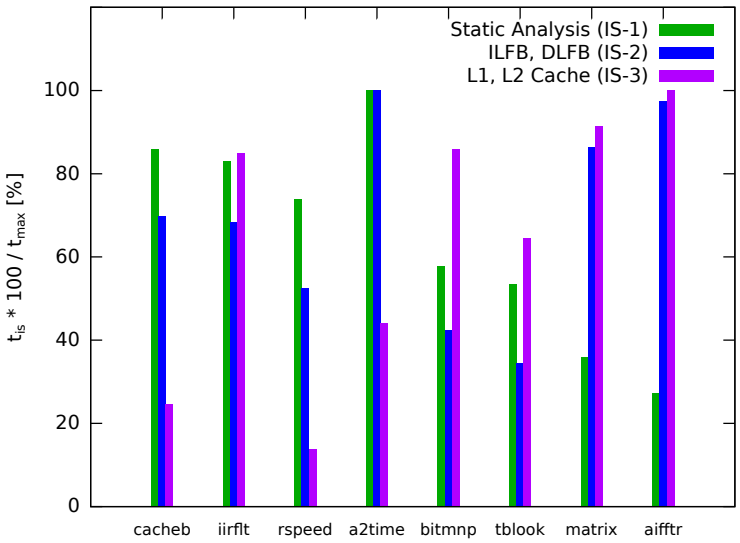


Figure 6.7.: *Relation of isWCET bounds t_{is} and maxCont t_{max} for Setups **IS-1**, **IS-2** and **IS-3**.*

The comparison of t_{is} and t_{min} shows great deviations to more than 1000%. Hence, from a performance perspective, the achieved multi-core bounds cannot compete with their single-core counterparts. Portions of the deviation can be explained by the already quantified overestimation of the core-local analysis. Especially the overestimation of the resource capacities has a great impact on the resulting isWCET bounds, considering the increase of the access latencies from 41 to 1007 cycles, cf. Table V. However, even if the tightness of the core-local analysis is increased, the deviation of t_{is} and t_{min} is up to 700%, as indicated by the measured core-local values with enabled L1 and L2 caches, shown in Table XV.

6.6. Quality of Service Monitoring Extension

As indicated by the comparison of the isWCET bounds and the minCont, the isWCET bounds incorporate significant overestimations. Table XVI complements these observation by comparing the single-core and the multi-core overestimation for the selected benchmarks. For that purpose, the table lists the statically analysed timing bounds t_{s,p_i}, t_{is} and maximum observed execution times t_{obs} and their deviation for single-core and multi-core execution. The observed execution times have been measured with disabled caches, i.e. Setup **IS-2**. The computed multi-core bounds t_{is} are based on the statically analysed core-local results, cf. Table XIII.

Table XVI.: *Comparison of single-core and multi-core overestimation based on the measured bounds t_{obs} and the core-local WCET bound t_{s,p_i} and the isWCET bound t_{is}.*

Benchmark	Single-core			Multi-core		
	t_{obs} $[ms]$	t_{s,p_i} $[ms]$	Deviation [%] $\frac{t_{s,p_i} \cdot 100}{t_{obs}} - 100$	t_{obs} $[ms]$	t_{is} $[ms]$	Deviation [%] $\frac{t_{is} \cdot 100}{t_{obs}} - 100$
cacheb	372	424	14.0	2912	7500	157.6
iirflt	363	532	46.7	4771	9722	103.7
rspeed	450	905	101.2	4593	12751	177.6
a2time	68	162	138.2	753	2907	285.9
bitmnp	1082	2586	139.1	10253	27937	172.5
tblook	1341	2664	98.7	14668	29792	103.1
matrix	231	6451	2695.8	2201	43714	1886.1
aifftr	89	7882	8714.5	995	47210	4647.0

Except for aifftr, the overestimation for the multi-core bound is higher than for the single-core analysis. Again, the particular overestimation greatly depends on the benchmark characteristics and the assumed overlap with the off-core requests of in-parallel scheduled benchmarks. In general, the tendency towards increased overestimation for the multi-core bounds confirms the assumption, that the significant increase of access latencies, cf. Table V, causes additional overestimation of the multi-core bounds. To cope with the induced idle times and low utilisation of the processor, the QoS extension has been introduced. Its goal is to utilise the otherwise unused processing time by dynamic re-calculation of the resource usage bounds, cf. Section 4.4, to allow applications to continue execution even if they have already exhausted their initial resource capacity. In this section the actual effect of this extension shall be characterised. Therefor, a set of application scenarios is defined based on the benchmark characterisation in Section 6.2:

Low-progress (lowp) Scenario: The lowp scenario shall contain a very limited extension potential. Therefor, a single benchmark is enabled for extensions. Its core-local execution time shall be as high as possible, in order to consume a large portion of the process frame with its first execution, i.e. without an extension. Thus a benchmark with a low locality is selected. To minimise the length of the process frame, benchmarks with a high locality are executed in-parallel. Furthermore, the effect of a capacity extension is minimised, since the higher the locality, the lower the initial resource capacities and the lower the potential progress compared to low-locality benchmarks.

Based on the benchmark classification, cacheb is selected as the capacity-extendible benchmark and multiple instances of bitmnp, tblook and matrix are scheduled on the remaining cores.

Realistic (real) Scenario: The real scenario shall represent a realistic task set. It contains a mixture of capacity-extendible and normal benchmarks, based on the application domains in Table VIII. Applications such as image and audio processing, and optimisation-based algorithms

are considered to benefit from additional execution time, cf. Section 4.4.3.

Accordingly, aifftr, matrix, iirflt, tblook and cacheb are enabled for capacity extensions, while rspeed, a2time and bitmnp will be suspended once they reach their initial resource capacity.

It has to be noted, that it is hard to prove if a task set is representative for future system. Firstly, today's avionics and automotive systems are based on single-core processors. Hence, the assignment of applications and cores as well as the configuration for the system can only be estimated. Secondly, application behaviour, such as cache usage, might change drastically in the future to cope with the contention on shared resources. However, since some future application domains can be foreseen, the constructed scenario is considered to adequately represent such a task set.

Both scenarios are parametrised once with statically analysed and once with measured core-local bounds. The measured bounds are acquired with either enabled L1 or with enabled L1 and L2 caches. The respective configurations are referred to as **lowp-sa, real-sa, lowp-l1, real-l1, lowp-l1-l2** and **real-l1-l2**, respectively.

As described for the implementation of the QoS extension, cf. Section 5.3.3, the overhead for the computation has to be considered. Table XVII shows the overheads in execution time t and resource usage c for the inter-core communication (DBell) and the modified ISR. The overheads are shown for static core-local analysis and for measurement-based analysis with different cache configurations. The comparison to Table X shows the difference to the basic ISR, which directly suspends a process. As can be seen, the modified ISR requires significantly more execution time and off-core requests, but still the overhead is negligible in comparison to the timing and resource bounds for the benchmarks.

Table XVII.: *Overhead for the inter-core communication (DBell) and the modified suspension ISR, showing the required bounds for execution time t and resource capacity c.*

	Static Analysis		Observed ILFB, DLFB		Observed L1 Caches		Observed L1, L2 Caches	
	t $[us]$	c	t $[us]$	c	t $[us]$	c	t $[us]$	c
DBell	10	243	4	73	2	43	2	30
ISR	4284	94659	132	1568	39	433	30	216

The scenarios are evaluated towards the achieved core and system utilisation, as well as the additional amount of off-core requests compared to the initial capacity. All results represent pure application execution, i.e. any overheads in time and resource usage for the computation of the capacity extensions and for taking measurements are discarded. The results for each scenario are summarised within two tables. The first table lists the applied core-local execution time bound t_{s,p_i} and resource capacity c_{p_i}. The resulting process frame length is shown in the caption. Further, the execution time and the related core utilisation are listed with and without extension $t_{ext}, t_{\overline{ext}}$ and $u_{ext}, u_{\overline{ext}}$, respectively. The results without extension correlate to the first execution of a benchmark. Accordingly, the results with extension comprise the results over the complete execution. Based on the core utilisations of all benchmarks, the overall system utilisation is computed. To quantify the benefit of the capacity extension, the relations of the utilisations with and without extension are shown. In the second table, the initial and the totally observed off-core requests c_{init} and c_{total} as well as their relation are listed. Both tables only list the observed minimum and maximum values for core-utilisation and capacity increase. However, the full results are listed in Appendix A.

Tables XVIII and XIX show the results for configuration **lowp-sa**. Since cacheb is the only

benchmark which is enabled for capacity extension, it executes for the complete length of the process frame, while the other benchmarks finish rather early. The large capacity extension of roughly 10^9 is explained by the relatively long process frame of $57s$ and the huge differences for the access latencies depending on the number of cores. That is, once the benchmarks on cores one to seven finished execution, all further accesses of cacheb can be accounted with the access latency for a single active core, i.e. $41cycles$ instead of $1007cycles$ otherwise. The relatively low system utilisation of 15% is expected, since the cores one to seven are idle most of the time.

Table XVIII.: *Utilisation for **lowp-sa** (process frame $57s$), showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

	Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
Min/Max	cacheb	424	9.9	632	57382	1.1	99.9	90.8
				System Utilisation		2.7	15.0	5.6

Table XIX.: *Additional off-core requests for **lowp-sa**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

	Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
Min/Max	cacheb	9.9	1184.5	119.6

The results for **lowp-l1**, are summarised in Tables XX and XXI. As can be seen, even though the core-local results are much tighter compared to the actual execution, the achieved core utilisation for cacheb is similar to the **lowp** scenario based on static analysis inputs. Solely the factor of additional requests is reduced to 5 compared to 120, comparing Tables XXI and XIX.

Table XX.: *Utilisation for **lowp-l1** (process frame $674ms$), showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

	Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
Min/Max	cacheb	5	3.4	72	668	10.7	99.2	9.3
				System Utilisation		20.2	31.3	1.5

Table XXI.: *Additional off-core requests for **lowp-l1**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

	Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
Min/Max	cacheb	3.4	17.2	5.1

The results for the **real** scenario based on statically analysed inputs are shown in Tables XXII and XXIII. Respectively, the results for measured inputs (L1 caches) are listed in Tables XXIV

and XXV. As for the **lowp** scenario, the characteristics of the results for the applied core-local analysis techniques are comparable. That is, the core utilisation for the benchmarks which are enabled for capacity extensions could be increased up 99.9%. Also the overall system utilisation is increased over 50%. The number of additional off-core requests differs, but overall each benchmark got at least twice the number of requests compared to its initial configuration. Further the increase factors are similar for the applied core-local analysis techniques, only their distribution over the benchmarks is different.

Table XXII.: *Utilisation for **real-sa** (process frame 47s), showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

	Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
Min	cacheb	424	9.9	724	39149	1.5	82.9	54.1
Max	aifftr	7882	197.8	239	47190	0.5	99.9	197.4
				System Utilisation		2.5	55.0	22.0

Table XXIII.: *Additional off-core requests for **real-sa**, showing the initial and totally measured capacities c_{init} and c_{total}.*

	Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
Min	matrix	137.4	359.0	2.6
Max	cacheb	9.9	463.6	46.8

Table XXIV.: *Utilisation for **real-l1** (process frame 1234ms), showing the core-local WCET bound t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

	Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
Min	cacheb	5	3.4	166	994	13.5	80.6	6.0
Max	iirflt	17	2.7	147	1223	11.9	99.2	8.3
				System Utilisation		10.2	61.5	6.0

Table XXV.: *Additional off-core requests for the **real-l1**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

	Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
Min	cacheb	3.4	12.7	3.8
Max	matrix	0.2	6.0	25.4

So far the results indicate significant benefits due to the QoS extension. However, with both, static and measurement-based (L1 caches) core-local analysis techniques all benchmarks have a relatively large number of off-core memory requests. To also assess task sets with rather low number

of requests, Tables XXVI and XXVII list the results for the **real** scenario and a measurement-based core-local analysis with enabled L1 and L2 caches. As can be seen, the number of off-core requests and the resulting process frame length are noticeably smaller than for the previous settings. This is explained by the relatively small code and data size of most benchmarks, such that they fit into the L2 caches. However, the achieved utilisation increases are still similar to those for static and measurement-based (L1 caches) core-local analysis methods.

Table XXVI.: *Utilisation for **real-l1-l2** (process frame* $612ms$*), showing the core-local WCET bound* t_{s,p_i} *and the capacity* c_{p_i}*; the observed execution times and, core and system utilisations with and without extension* $t_{ext}, t_{\overline{ext}}, u_{ext}$ *and* $u_{\overline{ext}}$*.*

	Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^3]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
Min	iirflt	57	94.3	62	486	10.1	79.6	7.9
Max	matrix	20	16.8	20	608	3.3	99.5	30.4
	aifftr	10	0.9	0	591	0.0	96.6	inf
					System Utilisation	9.5	64.2	6.7

Table XXVII.: *Additional off-core requests for **real-l1-l2**, showing the initial and the totally measured capacities* c_{init} *and* c_{total}*.*

	Benchmark	c_{init} $[10^3]$	c_{total} $[10^3]$	$\frac{c_{total}}{c_{init}}$
Min	matrix	0.0	0.1	3.3
Max	aifftr	0.0	0.3	285.0
	tblook	0.4	0.2	0.6
	cacheb	2.3	0.0	0.0

Considering the results in Table XXVI, it appears that the aifftr benchmark finishes after $0us$, causing an infinite utilisation increase. This is explained by the interval used to track the resource usage, i.e. the benchmark finished before the first interval. Similarly, in Table XXVII the observed numbers of off-core requests for tblook and cacheb are smaller than the statically assigned ones. Again, this is due to the benchmarks finishing their first execution before exhausting their capacity, while not obtaining a capacity extension afterwards.

7. Discussion

In this chapter the outcome of the thesis is reviewed. In Section 7.1 the benefits of the isWCET analysis and the runtime mechanism are discussed based on the evaluation results. The objectives defined in Chapters 1 and 3 are reviewed against the implementation and the evaluation results in Section 7.2. In Section 7.3, the validity of the assumptions made throughout the thesis are assessed and their impact on the generality of the work is discussed. Finally, deviations between the description of the concepts and the implementation are identified and discussed in Section 7.4.

7.1. Evaluation Results

7.1.1. Architecture Analysis

The main purpose of the architecture analysis is to determine the parameters, that are required for the timing analysis. In the context of the isWCET analysis, the key parameters are the access latencies for the off-core memory. They have been determined based on a specific interference application, sint, targeted towards worst-case interference on the NoC. The obtained latencies confirm the assumptions on significantly increased latencies with the number of concurrent cores and back up the abstraction of access latencies as defined in Section 4.3.2. The comparison of the latency for a single core (41 cycles) and eight cores (1007 cycles) indicates the pessimism imposed by the common assumption of a single, maximum access latency as for instance considered in [Schliecker et al. (2010), Yun et al. (2012), Behnam et al. (2012)]. Further, the measured relative increase renders the assumption of a linear increasing latency, as in [Dasari et al. (2011), Dasari and Nelis (2012)], incorrect.

Additionally, the impact of DMA-I/O devices has been examined. The results indicate a similar effect as for PEs, complementing the observations in [Schoenberg (2003)]. As a consequence, also interferences due to DMA-I/O need to be considered during analysis, cf. Section 4.5.

7.1.2. Resource Usage Enforcement

The implementation of the resource usage enforcement has been proven by executing a benchmark in isolation, with interference by other benchmarks and with interference but enabled partitioning. The results show that the mechanism behaves as intended, i.e. it fulfils its purpose of a safety-net with respect to the isolation of the resource requests of applications. The results also demonstrate the associated overheads for triggering and executing the suspension routine. The analysed and measured overheads for the suspension routine are negligible in comparison to the respective benchmarks values.

7.1.3. Core-local Analysis

The core-local analysis results have been evaluated based on their overestimation, comparing statically analysed and measured bounds for the core-local WCET and WCRA. As expected, the

individual benchmark characteristics, e.g. code structure, algorithm and optimisation for cache usage, heavily influence the absolute results. Since timing and resource analysis rely on the same techniques, the overestimation for both values is within the same order of magnitude. Overall, the results indicate the impact of the prototypical architecture model used for static analysis.

7.1.4. Interference-sensitive Analysis

To assess the benefits of the isWCET analysis, the computed bounds are compared against the minCont and maxCont approaches. The achieved reduction, of up 86.3% compared to maxCont proves the necessity and the benefits of the isWCET analysis. As for the core-local analysis, the benchmark characteristics influence the absolute values. Additionally, also the relative off-core access latencies impact the results. In particular, the larger the difference of the access latencies and WCRA bounds of in-parallel scheduled benchmarks, the higher the potential reduction. In practice, this can be used to optimise the system schedule, e.g. by combining an application with relatively few off-core requests with an application demanding a significantly larger amount.

To verify the impact of the prototypical architecture model, i.e. the missing cache levels, the isWCET bounds have been computed for different cache configurations and different core-local analysis methods. The comparison shows, that the absolute bounds heavily depend on the particular cache setup and the applied core-local analysis method, while the resulting reduction is comparable. Hence it can be concluded, that the benefits of the isWCET analysis are somewhat independent from the tightness of the core-local analysis.

Furthermore, comparing the sums over the minCont bounds of Tables XIII, XIV and XV ($38653ms$, $7196ms$, $744ms$) with the respective maximum isWCET bounds ($47210ms$, $9013ms$, $612ms$) shows that the parallel execution reduces the overall execution time only for Setup **IS-3**, shown in Table XV. Hence, from a performance perspective it can be concluded, that in most cases the analysed benchmarks do not benefit from the use of multi-core processors, when solely considering the worst-case. However, the actual effect depends on the application characteristics and in particular on the relation between core-local computation and off-core communication.

7.1.5. Quality of Service Monitoring Extension

The QoS extension is based on the assumption that the average-case behaviour of a system significantly deviates from its analysed worst-case configuration. Thus, the extension is designed to allow processes to utilise otherwise unused resources. Identified sources for the assumed deviations are the tightness of the core-local analysis results and overestimations in the isWCET analysis. Both effects have been verified during the evaluation. A second source of unused resources is, so-called growth potential. With respect to resources, it describes intentionally unused resources. They are reserved for later revisions of a system, that potentially implement additional applications and extended functionalities and thus require further resources. Since an extension of the hardware platform can be very costly, e.g. due to certification, the initial system design is overdimensioned. While in former system those resource cannot be used initially, by applying the QoS extension, they can be utilised even for earlier revision.

The evaluation of the QoS monitoring extension is based on the **lowp** and **real** scenarios, that have been constructed based on the benchmark characterisation. The results show a maximum system and core utilisation of 64.2% and 99.9%, respectively. The full utilisation of the core demonstrates the benefit of the QoS extension, especially when considering a maximum increase factor of 197.4 compared to the execution without any extension. It can also be seen, that the achieved maximum core utilisation of more than 99% is independent from the cache configuration

and the applied core-local analysis method. Similarly, the average core utilisations of capacity-extendible processes are close to equal, when comparing the different configurations of the **real**. In particular the following average core-utilisations have been observed for the different core-local analysis configurations: statically analysed 86.3%, measured-based with enabled L1 caches 90.9% and measurement-based with enabled L1 and L2 caches 94.9%. These observations conform with the results for the core-local analyses, where the absolute results heavily depend on the actual configuration, while the achieved benefits are rather independent from the configuration.

By definition the system utilisation is a function of the individual core utilisations. Hence, it depends on the combination of capacity-extendible and standard applications. This is confirmed by the comparison of the results for the **lowp** and the **real** scenario. The **lowp** configuration contains a single core with a capacity-extendible process, while the **real** scenario contains five respective processes. Accordingly, the maximum observed system utilisation for the **lowp** scenario is 31.3%, while it is 64.2% for the **real** scenario.

Finally, considering the additional number of off-core requests over all scenarios and configurations, a process obtains at least 2.6 times the number of requests compared to its initial configuration. This is true for each capacity-extendible process, except for tblook and cacheb, cf. Table XXVII, which finish execution before exhausting their initial bounds. The respective observed maximum increase is around 285.0 times.

Considering the benefits with respect to additional off-core requests and system utilisation, the QoS extension is considered as a key to increase the performance of a system, which is initially parametrised towards worst-case behaviour. Therefore, the QoS extension helps multi-core system to take advantage over single-core processors.

In summary, the validity and benefits of monitoring, isWCET analysis and QoS extension could be proven by the evaluation. The comparison of the results under different cache configurations and core-local analysis techniques demonstrates the independence of the benefits from the applied analysis method and the tightness of the initial application parameters. Instead, the benefits of the isWCET analysis and the QoS extension are impacted by two main factors:

1. Off-core Memory Access Latency Increase:
   The actual off-core latencies depend on the target platform. However, their increase with the number of PEs is an inherent problem of shared-memory multi-core architectures and thus a common phenomenon. Accordingly, once the problem of interfering requests has been solved on the architecture level, no additional mechanisms are necessary.
2. WCRA Bound Difference of In-Parallel Scheduled Processes:
   The tightness of the initial WCRA bounds depends on the application characteristics and the applied core-local analysis method. During the interference-delay analysis, the actual differences between the WCRA bounds determine the obtained reduction. Hence, depending on the tightness of the core-local analysis, the isWCET bound is directly influenced. However, during the evaluation it has been shown, that even for tightly analysed systems the QoS still provides reasonable performance improvements.

7.2. Thesis Objectives and Contributions

7.2.1. Timing bounds and Performance

The main quantitative objectives of this thesis are the reduction of multi-core timing bounds compared to straight forwards approaches and the efficient utilisation of multi-core processors in

order to take advantage over single-core platforms. As discussed in Section 7.1, the evaluation has proven significant reductions when using the isWCET analysis in favour of the maxCont approach, which is often inherently assumed in related approaches such as [Schliecker et al. (2010), Yun et al. (2012), Behnam et al. (2012)]. Further, also the increased average-case performance has been demonstrated in Section 6.6 and discussed in Section 7.1.

Unlike the related approaches of execution models, cf. Section 3.2, the presented mechanisms do not enforce resource privatisation by default. That means, that applications can utilise the full set of parallel resource. This avoids the inherent waste of performance, as for execution models, cf. Section 3.2. Additionally, the overheads for modifying the application binary are avoided, since the monitoring can be implemented based on available hardware features. With respect to certification, this specifically concerns overheads for modified compilers or additional tools, which need to fulfil safety requirements.

7.2.2. Mixed-criticality Systems

Another contribution is, to enable the analysis of mixed-critical systems, i.e. systems with applications of different assurance levels, cf. Section 2.2. Related approaches, e.g. [Yun et al. (2012), Paolieri et al. (2009)], often rely on the separation between hard real-time (critical) and non- or soft real-time (non-critical) applications, whereat critical applications are prioritised over non-critical applications, in terms of the resource usage. In effect, non-critical applications are executed under best-effort conditions, without any guarantees on the available resources. This contradicts the definition of mixed-criticality systems, where all applications are essential for the aimed system functionality and thus require guarantees on the available resources.

Neither the isWCET analysis nor the resource usage enforcement rely on information about the criticality of an application. That means, the runtime monitoring strictly enforces the adherence to the configured resource capacities. Hence, if a resource boundary does not safely upper-bound the resource usage of an application, the respective application will be suspended irrespective of its criticality. Accordingly, the core-local analysis has to be chosen such, that the obtained assurance is sufficient for the criticality level of the respective application. Hence, the isWCET analysis inherently considers the different implications of criticality levels through the choice of the applied core-local analysis technique. As described in Section 4.3.1, the core-local analysis can be implemented with any of the state of the art timing analysis techniques, and thus be chosen according to the required assurance. This has been proven during the evaluation, by applying static analysis as well as end-to-end measurements. Further, also hybrid measurement-based approach can be applied.

In consequence, the isWCET analysis and runtime resource usage enforcement fulfil the requirements of mixed-criticality systems.

7.2.3. Incremental Development and Certification

The applied analysis approach is separated into the core-local and the interference-delay analysis, cf. Section 4.3. Thus, the core-local timing bound and the resource usage of each application can be determined without requiring any information on other applications. Only during the interference-delay analysis the core-local bounds for WCET and WCRA of all applications are required to compute the respective process frame lengths. Thereby, the complexity of calculating the isWCET bounds is much lower than for the core-local analysis. Further, the impact on the system development time is smaller if a deviation from the defined parameters is detected earlier. That means, as described in Section 4.6, initial values the core-local timing bounds, the

WCRA bounds and also the system schedule are defined in the system design phase. During the actual implementation, each supplier can check the adherence to the defined bounds without requiring information on other applications. If a deviation is detected the implementation can immediately be revised. In contrast, for solutions that follow a joint analysis approach, deviations can only be detected during system integration, since the timing and resource usage bounds depend on the detailed behaviour of in-parallel scheduled applications. Accordingly, it takes much longer to detect deviations, slowing down development. For the same reasoning, modification to individual applications are much more costly if joint analysis approaches are applied. That is, if one application is modified, the impact of the modifications on other applications needs to be analysed. Following this discussion, the proposed isWCET analysis enables the independent analysis and development of applications and thus conform to the requirements of incremental development and certification.

An additional benefit compared to joint analysis approaches is the complexity. Joint analyses have to account for the mutual interactions between applications and thus face increased complexity with the number of concurrent applications. In contrast, the core-local analysis phase of the isWCET approach is based on state of the art single-core analysis techniques, without increasing the complexity. Final, the computation of the isWCET bound is of constant complexity for a given number of cores, while for joint analysis, the complexity depends on the state space of the individual application and thus is not constant, event for a given number of cores. Consequently, the objective towards reduced complexity compared to related approaches has been achieved.

7.3. Assumptions and Implementation

7.3.1. Operating System

In the implementation the operating system layer has been used to implement the resource usage enforcement. Therefor, the configuration of the PMCs and the respective ISR, implementing the suspension routine, have been added to the system. So far such implementations have been deployed to the bare-metal operating system, that has been used for the evaluation, SYSGO's PikeOS and Wind River's VxWorks. The variety of different operating systems demonstrates the independence of the monitoring solution from the underlying operating system. The operating system is required to provide interfaces to the PMCs and to the process management. For the implementation of the suspension routine, for instance a callback function, that is called upon a PMC exception can be used. An interface to the process management is required firstly, for the implementation of the suspension routine and secondly, for the configuration of the PMCs when a process is de/activated.

7.3.2. Monitoring Facility

The monitoring implementation is based on processor core PMCs. Such counters are commonly available in modern processors, e.g. [IBM (2010), ARM (2010), Intel (2013)]. Hence, this is not a very restrictive requirement. Further, the alternative platform debug facilities can be found in most modern SoC, cf. Section 4.2. As argued, due to the need for insight runtime information for system development, such debug architecture share a common set of features, which matches the requirements of the resource usage monitoring. However, if in a particular case neither processor core PMCs nor a sufficient debug architecture are available, also software-based monitoring based on code instrumentation can be applied.

Considering the available alternatives, it can be concluded, that the resource usage monitoring is a generally valid concept, not relying on a specific operating system or hardware platform.

7.3.3. Off-core Memory

The off-core memory resource is an abstraction of the memory architecture consisting of NoC, platform cache(s), memory controller and main memory, cf. Definition 5. Since it is essential for the execution of applications, the implementation and evaluation are focused on the off-core memory. For the analysis two main assumptions have been made:

Black-box Resource: This assumption is required, since often not enough details on the architecture and the implementation, especially of the NoC, are provided by the suppliers of COTS multi-core processors. To nevertheless perform an analysis, the black box approach in combination with am appropriate abstraction is reasonable. Consequently, the off-core memory is abstracted as its capacity. The capacity in turn, is determined via the access latencies in dependence of the number of requesters. While a purely measurement-based approach to acquire the respective latencies is sufficient for the evaluation of an research project, the methods for real products need to fulfil safety requirements. That is, the achieved assurance in acquiring informations on the resource need to be sufficient for the assurance level of the system. Consequently, it might be required, that the architecture and resource analysis needs to be based on detailed information on design and implementation, instead of pure measurements.

Fair Arbitration: Fair arbitration in any shared resource is a fundamental assumption for worst-case timing analysis. It ensures, that a starvation of individual requesters is not possible. This, in turn allows to determine upper bounds on access latencies. On the other hand, if a resource arbitration allows starvation, requesters might be not arbitrated for an indefinite amount of time. As a consequence, it is not possible to define a maximum waiting time, which prohibits worst-case timing analysis. Thus, the assumption of a fair arbitration is not exclusively required for the proposed approaches, instead it is motivated by the problem domain and likewise required for related approaches.

7.3.4. Timing Composability

In Section 2.4.3 the principles of timing composability and the impact of timing anomalies have been described. With respect to the isWCET analysis timing composability is required firstly, for the core-local analysis and secondly, for the composition of core-local results during the interference-delay analysis. The core-local analysis does not pose different assumptions or restrictions with respect to composability than existing approaches. On the other hand, since the interference-delay analysis is an additional step, its requirements for composability need to be considered separately. It requires composability to ensure, that the interference-delay is additive to the core-local WCET bound without impacting the correctness of the core-local analyses. This is argued to be the case, since naturally even the best-case off-core memory access latencies are orders of magnitudes higher than typical pipeline latencies. For example, typical pipeline latencies are between one to three cycles [FSL (2011)], while off-core memory latencies are measured between 39 and 1007 cycles, cf. Table V. As a consequence the processor pipeline will drain in any case, while an off-core request is processed. Thus the separate analysis of the interference-delay does not cause processor pipeline timing anomalies and hence can be directly added to the core-local WCET bound.

Even though the core-local analysis does not pose additional restrictions compared to existing approaches, its composability requirements need to be discussed in the context of the target

architecture. In general, timing analysis requires a predictable architecture, cf. [Cullmann et al. (2010)]. Unfortunately, according to [Wilhelm et al. (2009)], the P4080 has to be classified as a non-compositional architecture, since it implements potential sources of timing anomalies. This for instance concerns the out-of-order execution and the applied PLRU and First In First Out (FIFO) replacement schemes for caches, translation lookaside buffers, the branch target buffer and the branch history table of the e500mc cores. In order to be able to assume timing composability the cores can be used in a very deterministic configuration, which avoids any known sources of domino effects. Such a configuration includes disabled branch prediction, write-through mode for data caches, 2nd-level caches exclusively used as scratchpad memories, partial cache locking to achieve LRU replacement and pre-loaded translation lookaside buffers to prevent misses. Avoiding all these sources of domino effects still is does not formally prove timing composability. However, it shall be understood, that it is not easily possible to formally prove the absence of timing anomalies. Besides a predictable configuration also the means to handle anomalies, as described in Section 2.4.3, can be applied.

It is important to understand, that the impact of a non-compositional evaluation platform influences the safety of the bounds. However, this does not limit the validity of the approach, rather than the applicability of the architecture. While also a compositional architecture, such as the ARM7, could have been chosen, the P4080 has been selected for different reasons, as explained in Section 5.1. Further, missing composability does not prevent timing analysis, but greatly increase analysis complexity. For that purpose, composability is commonly assumed in related approaches, motivated by the problem domain to enable efficient analysis [Schliecker et al. (2010), Dasari and Nelis (2012)]. Finally, one should bear in mind, that for mixed-criticality systems the particular assurance level of an application has to be taken into account. For example, even if composability is not formally proven, it can still be valid to rely on the timing analysis, as long as the obtained results are suitable for the assurance level of the analysed application.

7.3.5. Interference-delay Analysis

The interference-delay analysis is based on the assumption, that the process frames are synchronised over all PEs. That means, process frame transitions happen at the point in time on all cores and the length of the respective process frames are equal in size. This assumption is directly deduced from the time partitioning scheme, described by ARINC 653 [ARINC (2003)]. Further, this assumption is inherently required for the computation of the isWCET boundaries. However, since a multi-core version of the ARINC 653 standard is not yet defined, it might be the case, that in an updated version of the standard, process frames are not synchronised over PE boundaries. In effect, it would be possible, that during a process frame of one PE another core executes two or more frames. Hence, the accesses of one process could overlap with the requests of two or more processes. To still be able to determine a isWCET boundary, the computation needs to be extended. Therefor, the following solutions are considered:

Split/Join of Process Frames: As indicated above, instead of overlaps with a single process per PE, processes can overlap with multiple processes. Accordingly, the sum over the requests of all processes that are scheduled in the same interval need to be considered during the computation. While this can easily be implemented, it causes an increased overestimation, since more requests within the same time interval can cause interference in the worst-case.

Fixed-size Monitoring Intervals: Instead of using the process frame granularity for the isWCET bound computation, a smaller fixed-size interval can be applied. Therefor, the core-local analyses need to be performed on application chunks of the size of that interval.

An inherent problem with this approach is the dependence between execution time and program

path. That means, depending on the analysed path the application has very different bounds for WCETs and WCRAs. Further, state of the art approaches for path analysis, e.g. ILPs, would need to be replaced, since they solve the maximum execution time problem for the whole set of paths, instead of terminating once a certain time is exceeded. To avoid these effects, it would help to a priori split applications. Hence, an application would be executed as a sequence of chunks, whereat each of the chunks is analysed separately. However, the general isWCET approach would still hold, while the application design would cover the division of the applications.

Reflecting the previous discussion it should be clear, that the current isWCET approach might not completely fulfil the requirements of future systems, but still its fundamental ideas can be applied.

In summary, the validity of the assumptions has been shown. The discussion revealed, that most of the assumptions are motivated by the problem domain and thereby are not more restrictive than for related approach. This specifically concerns assumptions with respect to analysability. Furthermore, the independence of isWCET analysis and runtime resource usage enforcement from unique hardware features and a specific operating system has been demonstrated.

7.4. Shortcomings

7.4.1. Architecture Model

The shortcomings of the architecture model applied for static analysis of the core-local behaviour has already been described in Section 5.4.1. The most severe difference to the e500mc cores is the memory hierarchy. While the e500mc cores implement three cache levels with ILFB, DLFB, L1 caches and L2 cache, the architecture model does only provide a single cache level. The impact on the analysis results has been investigated in Section 6.4.1, illustrating its severity.

The main effect of the deviation between core implementation and the model are significant overestimations of timing and resource usage. However, since the results based on measurements show similar results it can be concluded, that the model is sufficient for the evaluation, demonstrating the applicability of static analysis. As for the discussion of timing composability in Section 7.3, it should be understood, that the development of an accurate architecture model is extremely time-consuming. However, to apply the tool for the verification of safety-critical systems a model refinement and additional tool validation is required.

7.4.2. Differentiation of Read and Write Requests

As described in Chapter 5, read and write requests from the PEs to the off-core memory are not distinguished. The main effect of this is an additional overestimation, since the measured request latencies for read instructions are considerably lower than for write requests, cf. Table V. Hence, to obtain safe bounds, the respective maximum observed latency is applied. Thereby, the latencies some requests is overestimated.

The reasons for the missing differentiation vary depending on the analysis method. The static analysis does not provide the differentiation due to the prototype status of the analysis. That means, although the required information to compute separate WCRA bounds for read and write requests are available, it is not yet implemented. In the case of measurement-based analysis a differentiation is not implemented, since the available PMC events of the e500mc cores do not

allow to separately count read and write requests to the bus interface. Nevertheless, it might be possible to combine different events to indirectly acquire the desired values, e.g. the number of translated or completed load and store instructions. However, in order to retain a similar basis for the comparison of static and measurement-based analyses, the BIU event is commonly used.

7.4.3. Off-core Memory Abstraction

As described all parts of the off-core memory hierarchy are abstracted as a single resource. In effect it is not possible to distinguish access to individual parts, such as platform cache or main memory. In effect, since the access latencies can be very different, this causes additional overestimation in the computed isWCET bound.

To implement a more fine-grained approach both, analysis and runtime monitoring need to be extended. For static timing analysis the architecture needs to be modeled in more details, e.g. by adding the platform cache. To ease the analysis of shared caches techniques, such as hardware-supported or software cache partitioning should be applied, in order to avoid multiple PEs mutually evicting cache lines. Considering the complexity of multi-level caches, the required modeling effort and analysis complexity cannot easily be approximated. For measurement-based analysis, the monitoring facility has to be capable of differentiating between individual parts of the off-core memory. Typically, this requires SoC-level mechanisms, since PEs are not able to predetermine the target of a memory request as long as they are not directly addressable. The same applies to the resource usage monitoring.

7.4.4. Stripped Timing Analysis

Another source of overestimation is the core-local WCET bound. As described in Section 5.4.1, the static core-local timing analysis does not strip the off-core access latencies. Hence, the core-local WCET bound contains request latencies for all off-core requests, while the interference-delay analysis additionally accounts for request latencies. Since the computed WCRA bound can be very different from the number of off-core requests on the worst-case timing path, the additional latencies cannot easily be stripped. Instead the timing analysis needs to be extended to also count the number of off-core requests on the worst-case timing path.

On the other hand, such a mechanism has been implemented for the end-to-end measurements, proving the general feasibility.

7.4.5. Outstanding Transactions

Outstanding transactions describe load/store type instructions, that are currently issued to the core pipeline but not yet finished. They also describe instructions, that will occur due to write-back operations when cache lines are evicted. Those transactions are of special interest when suspending a process. Since the suspension has to ensure that this process does not issue any further requests to the off-core memory, it has to consider outstanding transactions. To handle in-fly operations in the pipeline, pipeline flush operations can be applied. To deal with potential future cache evictions a cache invalidation, which clears the valid bits of all cache lines without updating main memory, can be used. Since the suspension of a process is considered as a faulty state, the loss of the memory content is arguable.

Beneath the runtime control, the outstanding transaction also need to be considered during analysis. In particular, the worst-case number of load/store operations, that can be active in the pipeline once a suspension is triggered has to be analysed. Finally, this value has to be added

to the WCRA bound of the respective application. Similarly, write-back operations due to cache eviction can be handled, i.e. instead of invalidating the cache content, the maximum number of potential write-backs can be predetermined and added to the WCRA bound.

As mentioned in Section 5.3, outstanding transactions are not handled in the current implementation. Since this also impacts the safety of the system, it has to implemented for productively deployed systems but does not impact the presented results and conclusions.

In summary, this section covered different deviations between the described concepts and the implementation. During the discussion the following sources of overestimations have been identified:

- missing cache levels in architecture model, applied for static analysis,
- missing differentiation between read and write instructions during resource analysis,
- unstripped core-local WCET bounds, containing additional request latencies and
- abstraction of platform cache and main memory as one resource, disregarding the different access latency.

Other implementation shortcomings solely impact the safety of the computed bounds and hence need to be considered for productive systems. Nevertheless, it can be summarised, that none of the discussed shortcomings lowers the identified benefits or violates the conclusions.

8. Summary and Future Work

8.1. Summary and Conclusion

This thesis addressed the problems for the WCET analysis of multi-core processors due to inter-application interferences on implicitly shared resources. These problems have been discussed in the context of mixed-criticality systems as they are used in the aerospace and automotive industries. This application domain is especially interesting due to its requirements towards temporal partitioning to ensure proper isolation of applications and enabled incremental development and certification.

Motivated by the drawbacks of related approaches, the main objectives of this thesis are the reduction of the multi-core timing bounds compared to straight forward approaches and the efficient utilisation of multi-core systems in order to take advantage over established single-core processors. Based on these objectives, the so-called isWCET analysis concept and an associated runtime resource usage enforcement have been developed. The isWCET analysis is a timing analysis, which allows the computation of multi-core timing bounds, while considering arbitrary interferences by in-parallel scheduled applications. The runtime monitoring provides an additional safety-net, ensuring that the inherent assumptions made during analysis, are maintained at runtime. While the isWCET analysis is focused on bounding the worst-case behaviour of applications, a supplemental QoS extension has been introduced to improve the average-case performance. Its purpose is the utilisation of otherwise unused platform resources. As such, the QoS extension is a key to increase the system utilisation and compensate various sources of overestimation, which are inherently coupled with worst-case timing analysis.

The concepts have been evaluated on a state of the art COTS multi-core platform, the Freescale P4080. Timing analysis and runtime monitoring have been implemented in software developed as part of this thesis as well as in a commercial analysis framework and operating systems. The evaluation of the isWCET analysis revealed a maximum reduction of the multi-core timing bound of 86.3% compared to straight forward maximum contention analysis approaches. Hence, the objective towards reduced multi-core bounds have been achieved. However, the comparison to respective single-core bounds showed a significant deviation of up to 1163.0%. This indicates the performance and thereby the utilisation impact, if worst-case behaviour is considered exclusively. To address this issue, a QoS extension has been proposed. The respective evaluation results show an increase of the processor core utilisation from 0.5% to 99.9%, achieving a total system utilisation of up to 64.2%, compared to 9.5% otherwise. The results clearly demonstrate the benefits of the approach and the fulfilment of the defined objectives towards increased system utilisation.

Other objectives of the thesis, such as reduced complexity compared to related approaches are inherently achieved by the applied analysis method. Instead of applying costly mutual analysis or modifications of the application code, the isWCET analysis is based on extensions of existing methods, without increasing the complexity compared to single-core analysis. Additionally, the per se privatisation of resources has been avoided, which is considered essential for the efficient usage of modern multi-core systems. This especially concerns the inherent parallelism of resources.

Further, the requirements towards mixed-criticality and incremental certification and develop-

ment are achieved firstly, via an abstraction of the application behaviour, which does not required detailed information on in-parallel scheduled applications and secondly, by the equal treatment of the application resource boundaries at runtime. That means, instead of prioritising applications with higher criticality, all applications underlie the same rules. This in turn enables the guarantee of deadlines and resource usage for all applications, instead of having guarantees for highly critical applications, while executing lower critical applications in a best-effort manner.

Beneath the performance-wise and functional aspects also the generality of the proposed approaches has been verified. In particular, the independence from the underlying computing platform and the operating system has been shown. Further, it is possible to apply any of the state of the art timing analysis techniques. While the evaluation is based on static analysis and end-to-end measurements, also hybrid approaches can be applied. This complements the applicability of isWCET analysis and runtime resource usage enforcement to mixed-criticality system and enables incremental development and certification.

From a system perspective, the proposed approaches are orthogonal to higher-level concepts, that ensure spatial and temporal partitioning. Such approaches for instance include the applied hardware configuration and the mapping from applications to cores and memory devices.

In summary, plenty of research towards integrating multi-core processors in real-time systems has been done. However, the results of this thesis show, that the strict focus on worst-case behaviour either yields highly overestimated timing bounds or, if techniques to avoid the concurrent usage of shared resources are applied, system capabilities are wasted, which significantly reduces the achieved performance. In particular the results show, that overly increasing shared resource access latencies cause an increased deviation between average-case and worst-case behaviour compared to single-core processors. Hence, instead of performance benefits, multi-core processors often yield a performance degradation when solely focusing on worst-case behaviour. Accordingly, it is concluded, that a paradigm shift from the sole focus on worst-case behaviour towards the additional consideration of the average-case performance is advisable. It has been shown, that this can be achieved, while maintaining guarantees for worst-case scenarios.

8.2. Future Work

The developed isWCET analysis and runtime approaches provide a basis for the analysis and the runtime integration of applications on multi-core processors. However, appropriate scheduling techniques, which consider the effect of different application characteristics on the resulting isWCET bounds and therewith on the process frame have not been discussed. Hence, this is an important next step towards practical deployment. Such a scheduling especially concerns the selection of applications, that shall be scheduled in parallel. Beneath the impact on the timing bounds the general timing requirements defined by the system functionality need to be considered.

As discussed in Section 7.3.5, the current computation of the isWCET bounds is based on a synchronised execution of process frames over all cores. This is valid, since yet no multi-core version of the underlying software standard is available. However, once such a standard has been developed, the impact of its scheduling scheme on the isWCET analysis has to be analysed, considering the suggested modification of Section 7.3.5.

Another drawback of the current implementation is the tightness of the core-local results obtained by static analysis. The different sources of overestimation have been discussed in Section 7.4. Even though they are not essentially required to guarantee the safety of the approach, they should be addressed for practical system, in order to achieve reasonable bounds, such that application timing requirements can be fulfilled.

Further, the QoS extension can be enhanced to provide even better performance. So far only the progress with respect to the shared resource usage is accounted for the re-calculation of the resource capacities, while the progress for computational instructions is not considered. However, additional performance counters can be used to monitor the number of completed instructions and account them with minimal pipeline latencies. Hence, during the re-calculation the elapsed computational instruction execution time can also be reduced. However, the achievable improvements depend on the relation of the core-local WCET bound and the interference-delay.

Bibliography

[Allen (1970)] F. E. Allen. *Control Flow Analysis. Symposium on Compiler Optimization*, pp. 1–19, 1970. doi:10.1145/800028.808479.

[ARINC (2003)] *ARINC 653: Avionics Application Software Standard Interface.* Aeronautical Radio Inc., 2003.

[ARM (2009)] ARM Ltd. *CoreSight Components Technical Reference Manual*, 2009.

[ARM (2010)] ARM Ltd. *Cortex-A8 Technical Reference Manual*, 2010.

[Arnold et al. (1994)] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. *Bounding Worst-Case Instruction Cache Performance. 15th IEEE Real-Time Systems Symposium (RTSS)*, pp. 172–181, 1994. doi:10.1109/REAL.1994.342718.

[ARTEMIS (2010)] ARTEMIS Joint Undertaking (JU). *Reduced Certification Costs Using Trusted Multi-core Platforms (RECOMP)*, 2010. `http://www.recomp-project.eu/`. (last accessed on 16th Jan. 2014).

[Atanassov et al. (2001)] P. Atanassov, R. Kirner, and P. Puschner. *Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. IEEE Real-Time Embedded Systems Workshop*, 2001.

[AUTOSAR (2013)] *AUTOSAR - Automotive Open System Architecture.* AUTOSAR, 2013. `http://www.autosar.org`. (last accessed on 16th Jan. 2014).

[Behnam et al. (2012)] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. *Multi-core Composability in the Face of Memory-bus Contention. 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, vol. 10, pp. 35–42, 2012. doi:10.1145/2544350.2544354.

[Bellosa (1997a)] F. Bellosa. *Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment.* Technical report, University of Erlangen, 1997.

[Bellosa (1997b)] F. Bellosa. *Memory Access - The Third Dimension of Scheduling.* Technical report, University of Erlangen, 1997.

[Bernat et al. (2002)] G. Bernat, A. Colin, and S. M. Petters. *WCET Analysis of Probabilistic Hard Real-Time Systems. 23rd Real-Time Systems Symposium (RTSS)*, pp. 279–288, 2002. doi:10.1109/REAL.2002.1181582.

[Bernat et al. (2003)] G. Bernat, A. Colin, and S. Petters. *pWCET: A Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems.* Technical report, 2003.

[Betts (2010)] A. Betts. *Hybrid Measurement-Based WCET Analysis Using Instrumentation Point Graphs.* PhD thesis, University of York, 2010.

[Binns (2001)] P. Binns. *A Robust High-performance Time Partitioning Algorithm: The Digital Engine Operating System (DEOS) Approach. 20th Digital Avionics Systems Conference (DASC)*, vol. 1, pp. 1B6/1–1B6/12, 2001. doi:10.1109/DASC.2001.963309.

[BMBF (2011)] Bundesministerium für Bildung und Forschung (BMBF). *Automotive, Railway and Avionics Multicore Systems (ARAMiS)*, 2011. `http://www.projekt-aramis.de/projekt.php`. (last accessed on 16th Jan. 2014).

[Boniol et al. (2012)] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti. *Deterministic Execution Model on COTS Hardware. 25st Conference on Architecture of Computing Systems (ARCS)*, pp. 98–110, 2012. doi:10.1007/978-3-642-28293-5_9.

[Burns and Edgar (2001)] A. Burns and S. Edgar. *Statistical Analysis of WCET for Scheduling. 22nd IEEE Real-Time Systems Symposium (RTSS)*, pp. 215–224, 2001. doi:10.1109/REAL.2001.990614.

[Butler and Finelli (1993)] R. W. Butler and G. B. Finelli. *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering*, vol. 19, pp. 3–12, 1993. doi:10.1109/32.210303.

[Chapman (1994)] R. Chapman. *Program Timing Analysis.* Technical report, University of York, 1994.

[Chattopadhyay et al. (2010)] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. *Modeling Shared Cache and Bus in Multi-cores for Timing Analysis. 13th Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2010. doi:10.1145/1811212.1811220.

[Chattopadhyay et al. (2012)] S. Chattopadhyay, C. L. Kee, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. *A Unified WCET Analysis Framework for Multi-core Platforms. 18th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 99–108, 2012. doi:10.1109/RTAS.2012.26.

[Colin and Puaut (2000)] A. Colin and I. Puaut. *Worst Case Execution Time Analysis for a Processor with Branch Prediction. Journal of Real-Time Systems*, vol. 18, 2000. doi:10.1023/A:1008149332687.

[Colin and Puaut (2001)] A. Colin and I. Puaut. *A Modular & Retargetable Framework for Tree-based WCET Analysis. 13th Euromicro Conference of Real-Time Systems (ECRTS)*, pp. 37–44, 2001. doi:10.1109/EMRTS.2001.933995.

[Colin et al. (2002)] A. Colin, G. Bernat, and U. Kingdom. *Scope-tree: A Program Representation for Symbolic Worst-Case Execution Time Analysis. 14th Euromicro Conference on Real-Time Systems*, pp. 50–59, 2002. doi:10.1109/EMRTS.2002.1019185.

[Cullmann et al. (2010)] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm. *Predictability Considerations in the Design of Multi-Core Embedded Systems. Conference on Embedded Real Time Software and Systems (ERTS)*, pp. 36–42, 2010.

[Dasari and Nelis (2012)] D. Dasari and V. Nelis. *An Analysis of the Impact of Bus Contention on the WCET in Multicores. 14th IEEE Conference on Embedded Software and Systems (HPCC-ICESS)*, pp. 1450–1457, 2012. doi:10.1109/HPCC.2012.212.

[Dasari et al. (2011)] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. *Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. 10th IEEE Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1068–1075, 2011. doi:10.1109/TrustCom.2011.146.

[Dasari et al. (2013)] D. Dasari, B. Akesson, M. A. Awan, and S. M. Petters. *Identifying the Sources of Unpredictability in COTS-based Multicore Systems. 8th IEEE Symposium on Industrial Embedded Systems (SIES)*, pp. 39–48, 2013.

[Davis (1958)] M. Davis. *Computability & Unsolvability.* McGraw-Hill Book Company, 1958.

[Deslauriers et al. (2006)] F. Deslauriers, M. Langevin, G. Bois, and Y. Savaria. *RoC: A Scalable Network on Chip Based on the Token Ring Concept. IEEE North-East Workshop on Circuits and Systems*, 2006. doi:10.1109/NEWCAS.2006.250915.

[Diemer and Ernst (2010)] J. Diemer and R. Ernst. *Back Suction: Service Guarantees for Latency-Sensitive On-Chip Networks. 4th ACM/IEEE Symposium on Network-on-Chip (NOCS)*, pp. 155–162, 2010. doi:10.1109/NOCS.2010.38.

[Dubout and Fleuret (2012)] C. Dubout and F. Fleuret. *Exact Acceleration of Linear Object Detectors. 12th European Conference on Computer Vision (ECCV)*, pp. 301–311, 2012. doi:10.1007/978-3-642-33712-3_22.

[Duranton et al. (2011)] M. Duranton, D. Black-Schaffer, S. Yehia, and K. De Bosschere. *Computing Systems: Research Challenges Ahead - The HiPEAC Vision 2011/2012*. Technical report, High Performance and Embedded Architecture and Compilation (HiPEAC), 2011.

[EASA (2011)] EASA. *Certification Memorandum - Development Assurance of Airborne Electronic Hardware*. Technical report, Software & Complex Hardware section, European Aviation Safety Agency (EASA), 2011.

[EASA (2012)] *EASA Annual Safety Review*. European Aviation Safety Agency (EASA), 2012.

[EC (2012)] EC. *Mixed Criticality Systems*. Technical report, European Commission, 2012.

[EEMBC (2013)] The Embedded Microprocessor Benchmark Consortium. *EEMBC AutoBench 1.1 Benchmark Software*, 2013. `http://www.eembc.org/benchmark/automotive_sl.php`. (last accessed on 16th Jan. 2014).

[Engblom and Ermedahl (2000)] J. Engblom and A. Ermedahl. *Modeling Complex Flows for Worst-Case Execution Time Analysis. 21st IEEE Real-Time Systems Symposium (RTSS)*, pp. 163–174, 2000. doi:10.1109/REAL.2000.896006.

[Engblom and Jonsson (2002)] J. Engblom and B. Jonsson. *Processor Pipelines and their Properties for Static WCET Analysis. 2nd Conference on Embedded Software (EMSOFT)*, pp. 334–348, 2002. doi:10.1007/3-540-45828-X_25.

[Ferdinand et al. (1997)] C. Ferdinand, F. Martin, and R. Wilhelm. *Applying Compiler Techniques to Cache Behavior Prediction. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS)*, 1997.

[Flynn (1972)] M. J. Flynn. *Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972. doi:10.1109/TC.1972.5009071.

[FOKUS (2010)] Fraunhofer-Institut für offene Kommunikationssysteme FOKUS. *Multicore-Architektur zur Sensor-basierten Positionsverfolgung im Weltraum (MUSE)*, 2010. `http://www.fokus.fraunhofer.de/go/muse`. (last accessed on 16th Jan. 2014).

[FSL (2006)] Freescale Semiconductor. *e600 PowerPC Core Reference Manual*, Rev. 0, 2006.

[FSL (2011)] Freescale Semiconductor. *e500mc Core Reference Manual*, Rev. 0, 2011.

[FSL (2012)] Freescale Semiconductor. *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual*, 2012.

[Fuchsen (2010)] R. Fuchsen. *How to Address Certification for Multi-core Based IMA Platforms: Current Status and Potential Solutions. 29th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pp. 5.E.3–1–5.E.3–11, 2010. doi:10.1109/DASC.2010.5655461.

[Gallo et al. (2012)] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy. *Performance Evaluation of the Random Replacement Policy for Networks of Caches. 12th ACM SIGMETRICS/PERFORMANCE Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 395–396, 2012. doi:10.1145/2254756.2254810.

[GE (2009)] GE Aviation. *GE Aviation Provides Advanced Systems on the Boeing 787 Dreamliner First Flight*, 2009. `http://www.geaviation.com/press/systems/systems_20091215.html`. (last accessed on 23rd Jan. 2014).

[Gerdes et al. (2012a)] M. Gerdes, F. Kluge, T. Ungerer, and C. Rochange. *The Split-Phase Synchronisation Technique: Reducing the Pessimism in the WCET Analysis of Parallelised Hard Real-Time Programs. 18th IEEE Conference on Embedded and Rea-Time Computing Systems and Applications (RTCSA)*, pp. 88–97, 2012. doi:10.1109/RTCSA.2012.11.

[Gerdes et al. (2012b)] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat. *Time Analysable Synchronisation Techniques for Parallelised Hard Real-Time Applications. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 671–676, 2012. doi:10.1109/DATE.2012.6176555.

[Green et al. (2011)] B. Green, J. Marotta, B. Petre, K. Lillestolen, R. Spencer, N. Gupta, D. O'Leary, J. Lee, J. Strasburger, A. Nordsieck, B. Manners, and R. Mahapatra. *DOT/FAA/AR-11/2 - Handbook of the Selection and Evaluation of Microprocessors for Airborne Systems.* Technical report, Federal Aviation Administration (FAA), 2011.

[Hardy et al. (2009)] D. Hardy, T. Piquet, and I. Puaut. *Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. 30th IEEE Real-Time Systems Symposium (RTSS)*, pp. 68–77, 2009. doi:10.1109/RTSS.2009.34.

[Hattendorf et al. (2012)] A. Hattendorf, A. Raabe, and A. Knoll. *Shared Memory Protection for Spatial Separation in Multicore Architectures. 7th IEEE Symposium on Industrial Embedded Systems (SIES)*, pp. 299–302, 2012. doi:10.1109/SIES.2012.6356601.

[Hayhurst et al. (2001)] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. *A Practical Tutorial on Modified Condition/Decision Coverage.* Technical report, National Aeronautics and Space Administration (NASA), 2001.

[Healy and Whalley (1999)] C. Healy and D. Whalley. *Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, pp. 79–88, 1999. doi:10.1109/RTTAS.1999.777663.

[Healy et al. (1999)] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. *Bounding Pipeline and Instruction Cache Performance. IEEE Transactions on Computers*, vol. 48, pp. 53–70, 1999. doi:10.1109/12.743411.

[HighRely (2009)] HighRely. *DO-178B Costs Versus Benefits.* Whitepaper, 2009. `http://www.cems.uwe.ac.uk/~ngunton/hrwp_do_178b_cost_benefit.pdf`. (last accessed on 16th Jan. 2014).

[Hopkins and McDonald-Maier (2006)] A. Hopkins and K. McDonald-Maier. *Debug Support for Complex Systems on-Chip: A Review. IEE Proceedings on Computers and Digital Techniques*, vol. 153, pp. 197–207, 2006. doi:10.1049/ip-cdt:20050194.

[IBM (2010)] International Business Machines Corporation. *Power ISA*, 2010.

[IEC (2010)] *Functional Safety of Electrical/Electronic/Programmable Electric Safety-related Systems.* International Electrotechnical Commission (IEC), 2010.

[IEEE-ISTO (1999)] *The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface.* IEEE-ISTO Nexus 5001, 1999.

[Intel (2013)] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Part 2*, 2013.

[ISO (2011)] *ISO 26262 - Road Vehicles - Functional Safety.* International Organization for Standardization (ISO), 2011.

[Jean et al. (2012)] X. Jean, M. Gatti, G. Berthon, and M. Fumey. *MULCORS - The Use of MULticore proCessORS in Airborne Systems.* Technical report, European Aviation Safety Agency (EASA), 2012.

[Kelter et al. (2011)] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. *Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds. 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 3–12, 2011. doi:10.1109/ECRTS.2011.9.

[Kelter et al. (2013)] T. Kelter, T. Harde, P. Marwedel, and H. Falk. *Evaluation of Resource Arbitration Methods for Multi-core Real-time Systems. 13th Workshop on Worst-Case Execution time Analysis (WCET)*, pp. 1–11, 2013.

[Kinnan (2009)] L. Kinnan. *Use of Multicore Processors in Avionics Systems and its Potential Impact on Implementation and Certification. 28th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pp. 1.E.4–1–1.E.4–6, 2009. doi:10.1109/DASC.2009.5347560.

[Kopetz (1997)] H. Kopetz. *Real Time Systems - Design Principles for Distributed Embedded Applications (page 34)*. Kluwer Academic Publisher, 1997.

[Laprie et al. (2004)] J.-C. Laprie, B. Randell, and C. Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, 2004. doi:10.1109/TDSC.2004.2.

[Li et al. (2005)] X. Li, T. Mitra, and A. Roychoudhury. *Modeling Control Speculation for Timing Analysis. Journal of Real-Time Systems*, vol. 29, pp. 27–58, 2005. doi:10.1023/B:TIME.0000048933.15922.f9.

[Li et al. (2009)] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. *Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. 30th IEEE Real-Time Systems Symposium (RTSS)*, pp. 57–67, 2009. doi:10.1109/RTSS.2009.32.

[Li and Malik (1995)] Y.-t. S. Li and S. Malik. *Performance Analysis of Embedded Software Using Implicit Path Enumeration. 32nd ACM/IEEE Design Automation Conference (DAC)*, pp. 456–461, 1995. doi:10.1145/217474.217570.

[Li-Guo et al. (2008)] Z. Li-Guo, D. Hui-Min, and H. Jun-Gang. *Fault-Tolerance Ring Network on Chip without Buffer. Conference on Computer Science and Information Technology (ICCSIT)*, pp. 67–71, 2008. doi:10.1109/ICCSIT.2008.19.

[Lim et al. (1995)] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. *An Accurate Worst Case Execution Timing Analysis for RISC. IEEE Transactions on Software Engineering*, vol. 21, pp. 593–604, 1995. doi:10.1109/32.392980.

[Littlefield-Lawwill and Kinnan (2008)] J. Littlefield-Lawwill and L. Kinnan. *System Considerations for Robust Time and Space Partitioning in Integrated Modular Avionics. 27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pp. 1.B.1–1–1.B.1–11, 2008. doi:10.1109/DASC.2008.4702751.

[Liu and Layland (1973)] C. L. Liu and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment. Journal of the ACM*, vol. 20, pp. 46–61, 1973. doi:10.1145/321738.321743.

[Liu (2010)] J. Liu. *Evaluating Standard-Based Self-Virtualizing Devices: A Performance Study on 10 GbE NICs with SR-IOV Support. IEEE Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, 2010. doi:10.1109/IPDPS.2010.5470365.

[Lundqvist and Stenström (1999)] T. Lundqvist and P. Stenström. *Timing Anomalies in Dynamically Scheduled Microprocessors. 20th IEEE Real-Time Systems Symposium (RTSS)*, pp. 12–21, 1999. doi:10.1109/REAL.1999.818824.

[Matsutani et al. (2008)] H. Matsutani, M. Koibuchi, D. F. Hsu, and H. Amano. *Three-Dimensional Layout of On-Chip Tree-Based Networks. International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN)*, pp. 281–288, 2008. doi:10.1109/I-SPAN.2008.39.

[Matsutani et al. (2009)] H. Matsutani, M. Koibuchi, Y. Yamada, D. F. Hsu, and H. Amano. *Fat H-Tree: A Cost-Efficient Tree-Based On-Chip Network. IEEE Transactions on Parallel and Distributed Systems*, vol. 20, pp. 1126–1141, 2009. doi:10.1109/TPDS.2008.233.

[Mitra et al. (2002)] T. Mitra, A. Roychoudhury, and X. Li. *Timing Analysis of Embedded Software for Speculative Processors. 15th Symposium on System Synthesis*, pp. 126–131, 2002. doi:10.1145/581199.581229.

[MMA (2010)] *MMA02: Military Aviation Authority Master Glossary.* Military Aviation Authority (MAA), vol. 3, 2010.

[Münch et al. (2014)] D. Münch, M. Paulitsch, and A. Herkesdorf. *Temporal Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using PCIe SR-IOV. Workshop on Dependability and Fault Tolerance (VERFE)*, 2014.

[Paolieri et al. (2009)] M. Paolieri, F. J. Cazorla, M. Valero, and G. Bernat. *Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. 36th International Symposium on Computer Architecture (ISCA)*, pp. 57–68, 2009. doi:10.1145/1555815.1555764.

[Park (1993)] C. Y. Park. *Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. Journal of Real-Time Systems*, vol. 5, pp. 31–62, 1993. doi:10.1007/BF01088696.

[Park and Shaw (1990)] C. Y. Park and A. P. Shaw. *Experiments with a Program Timing Tool Based on Source-Level Timing Schema. 11th IEEE Real-Time Systems Symposium (RTSS)*, pp. 72–81, 1990. doi:10.1109/REAL.1990.128731.

[PCI-SIG (2007)] *Single Root I/O Virtualization and Sharing 1.1 Specification.* Peripheral Component Interconnect Special Interest Group (PCI-SIG), 2007.

[Pellizzoni and Caccamo (2007)] R. Pellizzoni and M. Caccamo. *Toward the Predictable Integration of Real-Time COTS based Systems. 28th IEEE Real-Time Systems Symposium (RTSS)*, pp. 73–82, 2007. doi:10.1109/RTSS.2007.15.

[Pellizzoni et al. (2008)] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. *Coscheduling of CPU and I/O Transactions in COTS-based Embedded Systems. 29th IEEE Real-Time Systems Symposium (RTSS)*, pp. 221–231, 2008. doi:10.1109/RTSS.2008.42.

[Pellizzoni et al. (2011)] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. *A Predictable Execution Model for COTS-Based Embedded Systems. 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 269–279, 2011. doi:10.1109/RTAS.2011.33.

[Petters (2002)] S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Technische Universität München, 2002.

[Prisaznuk (1992)] P. J. Prisaznuk. *Integrated Modular Avionics. IEEE National Aerospace and Electronics Conference (NAECON)*, vol. 1, pp. 39–45, 1992. doi:10.1109/NAECON.1992.220669.

[Puschner and Koza (1989)] P. Puschner and C. Koza. *Calculating the Maximum Execution Time of Real-Time Programs. Real-Time Systems Journal*, pp. 159–176, 1989. doi:10.1007/BF00571421.

[Puschner and Schedl (1995)] P. Puschner and A. Schedl. *Computing Maximum Task Execution Times with Linear Programming Techniques.* Technical report, Technische Universität Wien, 1995.

[Puschner and Schedl (1997)] P. Puschner and A. Schedl. *Computing Maximum Task Execution Times - A Graph-Based Approach. Journal of Real-Time Systems*, vol. 13, pp. 67–91, 1997.

doi:10.1023/A:1007905003094.

[QNX (2013)] QNX Software Systems. *QNX Neutrino Realtime Operating System: System Architecture*, 2013. `http://www.qnx.de/developers/docs/6.5.0/index.jsp?topic=%2Fcom.qnx.doc.neutrino_sys_arch%2Fsmp.html`. (last accessed on 16th Jan. 2014).

[Ramaprasad and Mueller (2005)] H. Ramaprasad and F. Mueller. *Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS)*, pp. 148–157, 2005. doi:10.1109/RTAS.2005.12.

[Ramaprasad and Mueller (2006)] H. Ramaprasad and F. Mueller. *Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 71–80, 2006. doi:10.1109/RTAS.2006.14.

[Rapita (2014)] Rapita. *RapiTime Explained*. Whitepaper, 2014.

[Reichenbach and Wold (2010)] F. Reichenbach and A. Wold. *Multi-core Technology – Next Evolution Step in Safety Critical Systems for Industrial Applications? 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pp. 339–346, 2010. doi:10.1109/DSD.2010.50.

[Reineke et al. (2006)] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. *A Definition and Classification of Timing Anomalies. 6th Workshop on Worst-Case Execution Time (WCET) Analysis*, pp. 1–6, 2006.

[Rosen et al. (2007)] J. Rosen, A. Andrei, P. Eles, and Z. Peng. *Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. 28th IEEE Real-Time Systems Symposium (RTSS)*, pp. 49–60, 2007. doi:10.1109/RTSS.2007.24.

[RTCA (2000)] *DO-254/ED-80 - Design Assurance Guidance for Airborne Electronic Hardware*. Radio Technical Commission for Aeronautics (RTCA), 2000.

[RTCA (2005)] *DO-297/ED-124 - Integrated Modules Avionics (IMA) Development Guidance and Certification Considerations*. Radio Technical Commission for Aeronautics (RTCA), 2005.

[RTCA (2012)] *DO-178C/ED-12C - Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), 2012.

[Rushby (1981)] J. Rushby. *Design and Verification of Secure Systems. 8th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 15, pp. 12–21, 1981. doi:10.1145/800216.806586.

[Rushby (2000)] J. Rushby. *Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance*. Technical report, National Aeronautics and Space Administration (NASA), 2000.

[SAE (1996)] *ARP4761 (Aerospace Recommended Practice) - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Society of Automotive Engineers (SAE), 1996.

[SAE (2010)] *ARP 4754A (Aerospace Recommended Practice) - Certification Considerations for Highly Integrated or Complex Aircraft Systems*. Society of Automotive Engineers (SAE), 2010.

[Salloum et al. (2012)] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek. *The ACROSS MPSoC - A New Generation of Multi-Core Processors designed for Safety-Critical Embedded Systems. 15th Euromicro Conference on Digital System Design (DSD)*, pp. 105–113, 2012. doi:10.1109/DSD.2012.126.

[Schliecker et al. (2010)] S. Schliecker, M. Negrean, and R. Ernst. *Bounding the Shared Resource Load for the Performance Analysis of Multiprocessor Systems. Design, Automation & Test in*

Europe Conference & Exhibition (DATE), pp. 759–764, 2010. doi:10.1109/DATE.2010.5456951.

[Schoenberg (2003)] S. Schoenberg. *Impact of PCI Bus Load on Applications in a PC Architecture. 24th IEEE Real-Time Systems Symposium (RTSS)*, pp. 430–439, 2003. doi:10.1109/REAL.2003.1253290.

[Schranzhofer et al. (2009)] A. Schranzhofer, J.-j. Chen, and L. Thiele. *Timing Predictability on Multi-Processor Systems with Shared Resources. Workshop on Reconciling Performance with Predictability (RePP)*, 2009.

[Shaw (1989)] A. Shaw. *Reasoning About Time in Higher-Level Language Software. IEEE Transactions on Software Engineering*, vol. 15, pp. 875–889, 1989. doi:10.1109/32.29487.

[Souyris et al. (2005)] J. Souyris, E. Le Pavec, G. Himbert, V. Jegu, and G. Borios. *Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. 5th Workshop on Worst-Case Execution Time (WCET) Analysis*, pp. 21–24, 2005. doi:10.4230/OASIcs.WCET.2005.810.

[Stappert et al. (2001)] F. Stappert, A. Ermedahl, and J. Engblom. *Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 132–140, 2001. doi:10.1145/502217.502240.

[Staschulat and Ernst (2004)] J. Staschulat and R. Ernst. *Multiple Process Execution in Cache Related Preemption Delay Analysis. 4th ACM Conference on Embedded Software*, pp. 278–286, 2004. doi:10.1145/1017753.1017798.

[Suhendra et al. (2006)] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. *Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis. 43rd ACM/IEEE Design Automation Conference (DAC)*, pp. 358–363, 2006. doi:10.1145/1146909.1147002.

[SYSGO (2008a)] SYSGO AG. *Airbus selects SYSGO's PikeOS as DO-178B Reference Platform for the A350 XWB*, 2008. `http://www.sysgo.com/en/news-events/press/press/details/article/airbus-selects-sysgos-pikeos-as-do-178b-reference-platform-for-the-a350-xwb/`. (last accessed on 16th Jan. 2014).

[SYSGO (2008b)] SYSGO AG. *Rheinmetall Selects DO-178B Certifiable PikeOS from SYSGO for A400M Project*, 2008. `http://www.sysgo.com/en/news-events/press/press/details/article/rheinmetall-selects-do-178b-certifiable-pikeos-from-sysgo-for-a400m-project/`. (last accessed on 16th Jan. 2014).

[Tanenbaum (2007)] A. S. Tanenbaum. *Modern Operating Systems*, vol. 3. Prentice Hall Press, 2007.

[Theiling and Ferdinand (1998)] H. Theiling and C. Ferdinand. *Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Analysis. 19th IEEE Real-Time Systems Symposium (RTSS)*, pp. 144–153, 1998. doi:10.1109/REAL.1998.739739.

[Theiling et al. (2000)] H. Theiling, C. Ferdinand, and R. Wilhelm. *Fast and Precise WCET Prediction by Separated Cache and Path Analyses. Journal of Real-Time Systems*, vol. 18, pp. 157–179, 2000. doi:10.1023/A:1008141130870.

[Thesing et al. (2003)] S. Thesing, R. Heckmann, J. Souyris, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. *An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. 33th Conference on Dependable Systems and Networks (DSN)*, pp. 625–632, 2003. doi:10.1109/DSN.2003.1209972.

[Triquet (2012)] B. Triquet. *Mixed Criticality in Avionics. Workshop on Mixed Criticality Systems*,

pp. 1–7, 2012.

[Turing (1936)] A. M. Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society*, vol. 38, pp. 173–198, 1936. doi:10.1112/plms/s2-42.1.230.

[Udipi et al. (2010)] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian. *Towards Scalable, Energy-Efficient, Bus-Based On-Chip Networks. 16th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–12, 2010. doi:10.1109/HPCA.2010.5416639.

[Ungerer (1997)] T. Ungerer. *Parallelrechner und Parallele Programmierung.* Spektrum, Akademischer Verlag, 1997.

[Walter et al. (2008)] I. Walter, I. Cidon, and A. Kolodny. *BENoC: A Bus-Enhanced Network on-Chip for a Power Efficient CMP. Computer Architecture Letters*, vol. 7, pp. 61–64, 2008. doi:10.1109/L-CA.2008.11.

[Watkins and Walter (2007)] C. B. Watkins and R. Walter. *Transitioning from Federated Avionics Architectures to Integrated Modular Avionics. 26th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pp. 2.A.1–1–2.A.1–10, 2007.

[Wenzel et al. (2005)] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. *Principles of Timing Anomalies in Superscalar Processors. 5th IEEE Conference on Quality Software (QSIC)*, pp. 295–303, 2005. doi:10.1109/QSIC.2005.49.

[Wilhelm et al. (2008)] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, and J. Staschulat. *The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, pp. 1–53, 2008. doi:10.1145/1347375.1347389.

[Wilhelm et al. (2009)] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. *Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, pp. 966–978, 2009. doi:10.1109/TCAD.2009.2013287.

[Willmann et al. (2007)] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, and W. Zwaenepoel. *Concurrent Direct Network Access for Virtual Machine Monitors. 13th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pp. 306–317, 2007. doi:10.1109/HPCA.2007.346208.

[Wilson and Preyssler (2008)] A. Wilson and T. Preyssler. *Incremental Certification and Integrated Modular Avionics. 27th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pp. 1.E.3–1–1.E.3–8, 2008. doi:10.1109/DASC.2008.4702768.

[Wind River (2004)] Wind River. *Wind River Platform for Safety Critical ARINC 653 Selected by EADS/CASA - the Spanish Aerospace Division of EADS*, 2004. `http://www.windriver.com/news/press/pr.html?ID=201`. (last accessed on 23rd Jan. 2014).

[Yan and Zhang (2008)] J. Yan and W. Zhang. *WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 80–89, 2008. doi:10.1109/RTAS.2008.6.

[Yun et al. (2012)] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. *Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 299–308, 2012. doi:10.1109/ECRTS.2012.32.

[Yun et al. (2013)] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. *MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 55–64,

2013. doi:10.1109/RTAS.2013.6531079.

List of Figures

List of Tables

List of Acronyms

AMP	Asymmetric Multiprocessing.
ASIL	Automotive Safety Integrity Level.
AST	Abstract Syntax Tree.
BCET	Best-Case Execution Time.
BIU	Bus Interface Unit.
BMP	Bound Multiprocessing.
CFG	Control Flow Graph.
COTS	Commercial Off-The-Shelf.
CP	Constrained Program.
DAL	Design Assurance Level.
DC	Decision Coverage.
DEOS	Digital Engine Operating System.
DLFB	Data Line Fill Buffer.
DMA	Direct Memory Access.
DMA-I/O	DMA-capable Input/Output.
DPAA	Data Path Acceleration Architecture.
EASA	European Aviation Safety Agency.
EDF	Earliest Deadline First.
EPU	Event Processing Unit.
FAA	Federal Aviation Administration.
FAR	Federal Aviation Regulations.
FFT	Fast Fourier Transform.
FIFO	First In First Out.
FPGA	Field Programmable Gate Array.
GPOS	General-Purpose Operating System.
I^2C	Inter-Integrated Circuit.
I/O	Input/Output.
I/OMMU	Input/Output MMU.
ILFB	Instruction Line Fill Buffer.
ILP	Integer Linear Program.

IMA	Integrated Modular Avionics.
IPET	Implicit Path Enumeration Technique.
ISA	Instruction Set Architecture.
ISR	Interrupt Service Routine.
isWCET	interference-sensitive Worst-Case Execution Time.
JAR	Joint Aviation Regulations.
L1	level-1.
L2	level-2.
L3	level-3.
LRM	Line Replaceable Module.
LRU	Least Recently Used.
maxCont	maximum-Contention.
MC/DC	Modified Condition/Decision Coverage.
MIMD	Multiple Instruction Multiple Data.
minCont	minimum-Contention.
MMU	Memory Management Unit.
MPIC	Multi-core Programmable Interrupt Controller.
MPSoC	Multi-Processor System-on-Chip.
MPU	Memory Protection Unit.
NoC	Network-on-Chip.
OET	Observed Execution Time.
PAMU	Peripheral Access Management Unit.
PCI/PCIe	Peripheral Component Interconnect Express.
PE	Processing Element.
PLRU	Pseudo Least Recently Used.
PMC	Performance Monitor Counter.
PREM	PRedictable Execution Model.
QoS	Quality of Service.
RMS	Rate-Monotonic Scheduling.
RTOS	Real-Time Operating System.
SC	Statement Coverage.
SEU	Single Event Upset.
SIL	Safety Integrity Level.
sInt	synthetic Interference.

SK	Separation Kernel.
SMP	Symmetric Multiprocessing.
SoC	System-on-Chip.
SPI	Serial Peripheral Interface.
SRAM	Static Random Access Memory.
sRIO	serial RapidIO.
SWaP	Space, Weight and Power.
syscall	System Call.
TDMA	Time Division Multiple Access.
UART	Universal Asynchronous Receiver/Transmitter.
WCET	Worst-Case Execution Time.
WCRA	Worst-Case number of shared Resource Accesses.

A. Quality of Service Monitoring - Full Evaluation Results

The results for a scenario are summarised within a figure and two tables. The figure plots the observed off-core resource usage per core. The plot for each core shows the number of off-core requests over time, while suspension and capacity extension points are indicated as red triangles. It shall be noted, that plots for individual cores are cut off on the y-axis to increase the visibility of the data for other cores. The respective first table lists the applied core-local parameters for execution time bound t_{s,p_i} and resource capacity c_{p_i}, the process frame, the measured execution times and the resulting core and system utilisation. Execution times and utilisations are shown for the first execution of each benchmark $t_{\overline{ext}}, u_{\overline{ext}}$ and for the overall execution t_{ext}, u_{ext}. To quantify the benefit of the QoS extension, the relations between the utilisations with and without extension are computed. The second table compares the initial resource capacities c_{init} and the total amount c_{total}, which includes the sum over all capacity extensions for that benchmark. The listed results only contain process related execution time and requests, i.e. all overheads due to the execution of the suspension routine and the computation of the capacity extension are discarded from the results.

For the description of the measurement configurations refer to Section 6.6.

A.1. Scenario lowp-sa

Table XXVIII.: *Utilisation for **lowp-sa**, showing the core-local WCET bounds t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
cacheb	424	9.9	632	57382	1.1	99.9	90.8
bitmnp	2586	54.5	1923	-	3.4	-	-
tblook	2664	63.2	2451	-	4.3	-	-
matrix	6451	137.4	481	-	0.8	-	-
bitmnp	2586	54.5	1921	-	3.3	-	-
tblook	2664	63.2	2451	-	4.3	-	-
matrix	6451	137.4	478	-	0.8	-	-
bitmnp	2586	54.5	1924	-	3.4	-	-
				System Utilisation	2.7	15.0	5.6
				Process Frame [ms]	57397		

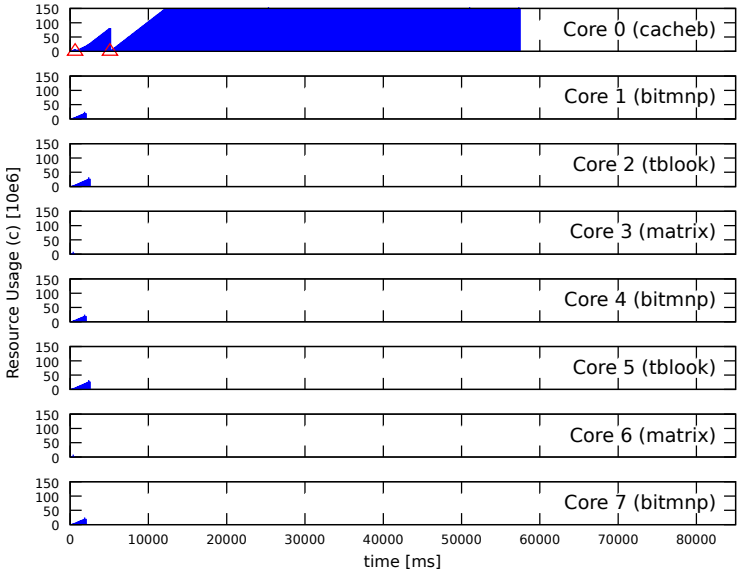


Figure A.1.: *Observed off-core memory requests c over time for **lowp-sa**.*

Table XXIX.: *Additional off-core requests for **lowp-sa**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
cacheb	9.9	1184.5	119.6

A.2. Scenario lowp-l1

Table XXX.: *Utilisation for* ***lowp-l1****, showing the core-local WCET bounds* t_{s,p_i} *and the capacity* c_{p_i}*; the observed execution times and, core and system utilisations with and without extension* $t_{ext}, t_{\overline{ext}}, u_{ext}$ *and* $u_{\overline{ext}}$*.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
cacheb	5	3.4	72	668	10.7	99.2	9.3
bitmnp	160	0.7	192	-	28.5	-	-
tblook	114	1.0	186	-	27.7	-	-
matrix	18	0.2	31	-	4.7	-	-
bitmnp	160	0.7	194	-	28.8	-	-
tblook	114	1.0	186	-	27.7	-	-
matrix	18	0.2	31	-	4.7	-	-
bitmnp	160	0.7	195	-	29.0	-	-
				System Utilisation	20.2	31.3	1.5
				Process Frame [ms]		674	

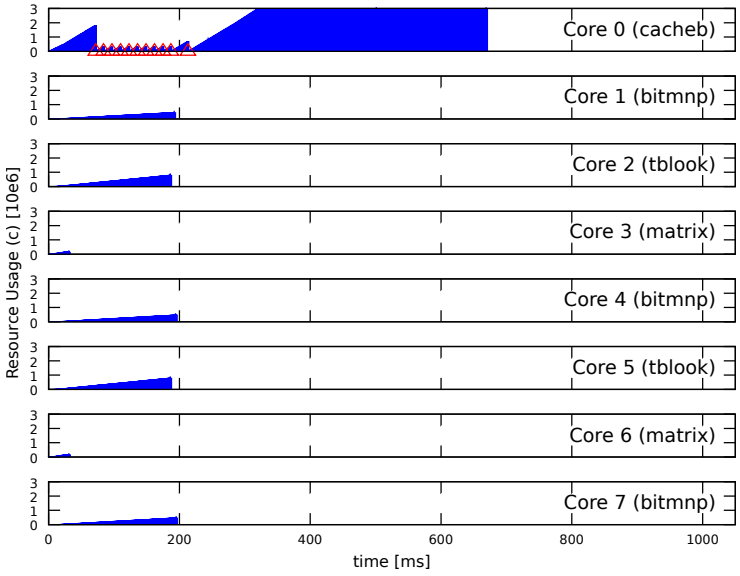


Figure A.2.: *Observed off-core memory requests c over time for* ***lowp-l1****.*

Table XXXI.: *Additional off-core requests for* ***lowp-l1****, showing the initial and the totally measured capacities* c_{init} *and* c_{total}*.*

Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
cacheb	3.4	17.2	5.1

A.3. Scenario real-sa

Table XXXII.: *Utilisation for **real-sa**, showing the core-local WCET bounds t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
aifftr	7882	197.8	239	47190	0.5	99.9	197.4
matrix	6451	137.4	567	39201	1.2	83.0	69.1
rspeed	905	19.5	1082	-	2.3	-	-
iirflt	532	13.3	894	39176	1.9	83.0	43.8
tblook	2664	63.2	3306	39222	7.0	83.1	11.9
a2time	162	3.3	167	-	0.4	-	-
bitmnp	2586	54.5	2458	-	5.2	-	-
cacheb	424	9.9	724	39149	1.5	82.9	54.1
			System Utilisation		2.5	55.0	22.0
			Process Frame [ms]		47210		

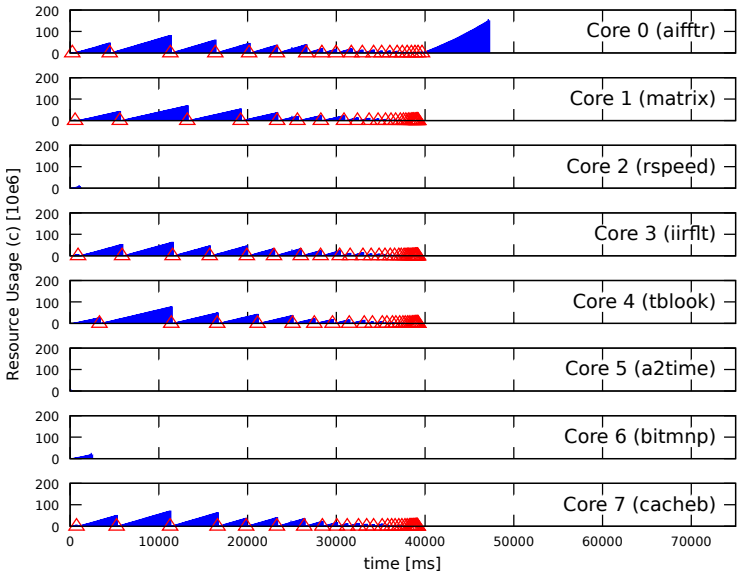


Figure A.3.: *Observed off-core memory requests c over time for **real-sa**.*

Table XXXIII.: *Additional off-core requests for **real-sa**, showing the initial and totally measured capacities c_{init} and c_{total}.*

Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
aifftr	197.8	633.2	3.2
matrix	137.4	359.0	2.6
iirflt	13.3	436.9	32.7
tblook	63.2	351.1	5.6
cacheb	9.9	463.6	46.8

A.4. Scenario real-l1

Table XXXIV.: *Utilisation for **real-l1**, showing the core-local WCET bounds t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^6]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
aifftr	7	0.4	50	1087	4.1	88.2	21.8
matrix	18	0.2	40	1206	3.2	97.8	30.2
rspeed	12	8.0	202	-	16.4	-	-
iirflt	17	2.7	147	1223	11.9	99.2	8.3
tblook	114	1.0	145	1096	11.8	88.9	7.6
a2time	4	0.8	45	-	3.7	-	-
bitmnp	160	0.7	207	-	16.9	-	-
cacheb	5	3.4	166	994	13.5	80.6	6.0
			System Utilisation		10.2	61.5	6.0
			Process Frame [ms]		1234		

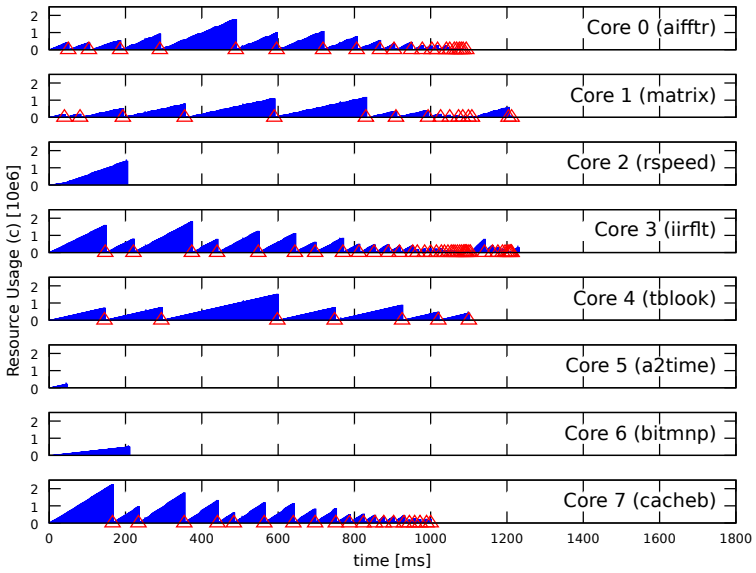


Figure A.4.: *Observed off-core memory requests c over time for **real-l1**.*

Table XXXV.: *Additional off-core requests for the **real-l1**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

Benchmark	c_{init} $[10^6]$	c_{total} $[10^6]$	$\frac{c_{total}}{c_{init}}$
aifftr	0.4	9.6	22.0
matrix	0.2	6.0	25.4
iirflt	2.7	14.9	5.4
tblook	1.0	4.9	4.7
cacheb	3.4	12.7	3.8

A.5. Scenario real-l1-l2

Table XXXVI.: *Utilisation for **real-l1-l2**, showing the core-local WCET bounds t_{s,p_i} and the capacity c_{p_i}; the observed execution times and, core and system utilisations with and without extension $t_{ext}, t_{\overline{ext}}, u_{ext}$ and $u_{\overline{ext}}$.*

Benchmark	t_{s,p_i} $[ms]$	c_{p_i} $[10^3]$	$t_{\overline{ext}}$ $[ms]$	t_{ext} $[ms]$	$u_{\overline{ext}}$ [%]	u_{ext} [%]	$\frac{u_{ext}}{u_{\overline{ext}}}$
aifftr	10	0.9	0	591	0.0	96.6	inf
matrix	20	16.8	20	608	3.3	99.5	30.4
rspeed	53	5261.2	75	-	12.4	-	-
iirflt	57	94.3	62	486	10.1	79.6	7.9
tblook	108	405.6	112	608	18.3	99.5	5.4
a2time	7	616.0	12	-	2.1	-	-
bitmnp	148	135.2	149	-	24.5	-	-
cacheb	33	2337.8	34	608	5.6	99.5	17.9
				System Utilisation	9.5	64.2	6.7
				Process Frame [ms]	612		

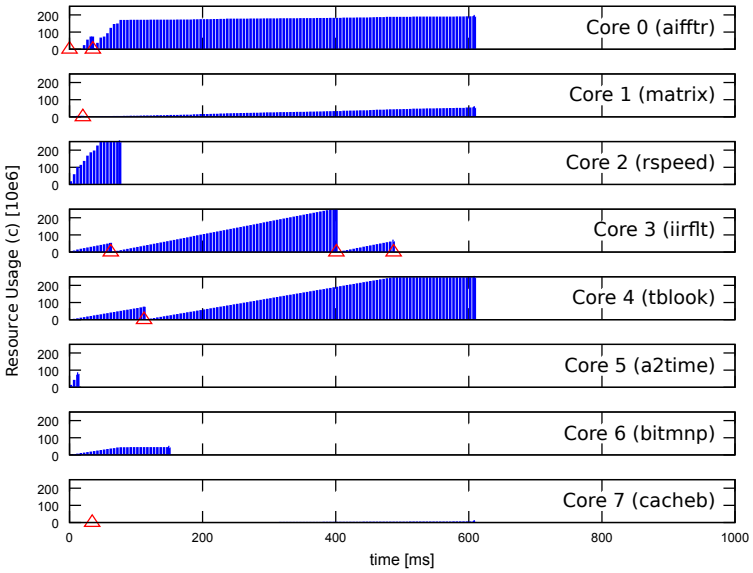


Figure A.5.: *Observed off-core memory requests c over time for **real-l1-l2**.*

Table XXXVII.: *Additional off-core requests for **real-l1-l2**, showing the initial and the totally measured capacities c_{init} and c_{total}.*

Benchmark	c_{init} $[10^3]$	c_{total} $[10^3]$	$\frac{c_{total}}{c_{init}}$
aifftr	0.0	0.3	285.0
matrix	0.0	0.1	3.3
iirflt	0.1	0.3	3.4
tblook	0.4	0.2	0.6
cacheb	2.3	0.0	0.0

Acknowledgements

At first I want to thank my advisers Prof. Theo Ungerer and Michael Paulitsch for their support and feedback during this thesis. Further I want to thank Prof. Theo Ungerer and Prof. Rudi Knorr for rating the thesis. I also want to acknowledge the work of Christian Ferdinand, Simon Wegener und Michael Schmidt from AbsInt, for their fruitful discussion and effort in extending their static analysis framework, which has been used for parts of the evaluation. Likewise, I want to thank Henrik Theiling from SYSGO for inspiring discussions. My special thanks goes to my colleagues Michael Paulitsch, Kevin Müller and Johannes Schels, as well as to my family for the continuous support, distraction and the joyful time.

Danksagung

Zunächst möchte ich meinen Betreuern Prof. Theo Ungerer und Michael Paulitsch für ihren Rat und ihre Hilfestellung während der Erarbeitung dieser Dissertation danken. Prof. Theo Ungerer und Prof. Rudi Knorr danke ich außerdem für die Bewertung der Arbeit. Weiterhin gilt mein Dank Christian Ferdinand, Simon Wegener und Michael Schmidt von der Firma AbsInt, für die interessanten Diskussionen und die Erweiterung ihres Frameworks für statische WCET Analyse, welches für Teile der Evaluierung verwendet wurde. Ebenso danke ich Henrik Theiling von SYSGO für anregende Diskussionen. Mein spezieller Dank geht an meine Kollegen Michael Paulitsch, Kevin Müller und Johannes Schels, sowie meine Familie für die fortwährende Unterstützung, Ablenkung und eine wirklich schöne Zeit.

Curriculum Vitae

Personal Information

Name:	Jan Nowotsch
Date of Birth:	14.06.1987
Place of Birth:	Karl-Marx-Stadt, Germany

Education

1993 - 1997:	Primary School at Obere Luisenschule Chemnitz
1993 - 2005:	Gymnasium at Gymnasium am Schlossteich, later Dr. Wilhelm André Gymnasium Graduation: Allgemeine Hochschulreife
2005 - 2010:	Study of Applied Computer Science at Technical University Chemnitz Graduation: Diploma
since 2010:	Ph.D. student at Airbus Group Innovations (former EADS Innovation Works) supervised at Department of Computer Science at University of Augsburg

Publications

Author

[1] J. Nowotsch and M. Paulitsch. *Leveraging Multi-Core Computing Architectures in Avionics. 9th European Dependable Computing Conference (EDCC)*, pp. 132–143, 2012. doi:10.1109/EDCC.2012.27.

[2] J. Nowotsch and M. Paulitsch. *Quality of Service Capabilities for Hard Real-Time Applications on Multi-core Processors. 21st Conference on Real-Time Networks and Systems (RTNS)*, pp. 151–160, 2013. doi:10.1145/2516821.2516826.

[3] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. *Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 109–118, 2014. doi:10.1109/ECRTS.2014.20. also available as technical report at `opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/year/2013/docId/2474`.

[4] J. Nowotsch, M. Paulitsch, A. Henrichsen, W. Pongratz, and A. Schacht. *Monitoring and WCET Analysis in COTS Multi-core-SoC-based Mixed-Criticality Systems. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014.

Associated Author

[1] O. Kotoba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. *Multi-Core In Real-Time Systems – Temporal Isolation Challenges Due To Shared Resources. Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (WICERT), as part of DATE conference*, 2013.

[2] M. Paulitsch, L. Girbinger, J. Nowotsch, and D. Münch. *Transparent Software Replication and Hardware Monitoring Leveraging Modern System-On-Chip Features. 19th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013.

[3] C. Sánchez, J. Nowotsch, M. Paulitsch, and K. Schertler. *Image Processing in Airborne Applications Using Multicore Embedded Computers. 32nd Digital Avionics Systems Conference (DASC)*, 2013.

Patents

[1] J. Nowotsch and M. Paulitsch. *Zugangssteuerung zu einem gemeinsamen exklusiv nutzbaren Übertragungsmedium*. Patent DE102011114378A1, 2011. `https://www.google.com/patents/DE102011114378A1?cl=de&dq=paulitsch+nowotsch&ei=QYD4UuSpKsfKtAauqIHADQ`. (not yet approved).

[2] M. Paulitsch, J. Nowotsch, L. Girbinger, D. Bühler, A. Herkesdorf, and T. Wild. *Redundantes Mehrprozessorsystem und zugehöriges Verfahren*. Patent EP2662773A1, 2012. `https://www.google.com/patents/EP2662773A1?cl=de&dq=paulitsch+nowotsch&ei=QYD4UuSpKsfKtAauqIHADQ`. (not yet approved).